# HiSoft FORTH

*the FORTH for your Atari ST Computer*

**Requires:**

- ✓ *520ST upwards*
- ✓ *Disk drive + mouse*

HiSoft
High Quality Software

# HiSoft FORTH

## for your Atari ST

**System Requirements:**
Atari ST Computer with a mouse and a disk drive

**Program Copyright © Henry McGeough 1988-90**
**Manual Copyright © Henry McGeough and HiSoft1990**

**HiSoft FORTH for the Atari ST, February 1990**

**Printing History:**
**1st Edition** February 1990 (ISBN 0 948517 24 7)

Set using an Apple Macintosh™ with Microsoft Word™ & Aldus Pagemaker™

ISBN 0 948517 24 7

All Rights Reserved Worldwide. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, without the written permission of the copyright holder. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature.

It is an infringement of the copyright pertaining to **HiSoft FORTH** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft FORTH** for any reason other than for the purposes of making a security back-up copy of the object code as detailed within this manual.

# HiSoft FORTH
# Table of Contents


**1 Introduction 1**

1.1 FORTH History 1
1.2 About HiSoft FORTH 1
1.3 Acknowledgements 2
1.4 How to use this manual 2
1.5 Always make a back-up 2
1.6 Registration Card 3
1.7 HiSoft FORTH Disk Contents 4
1.8 The README File 5
1.9 Making a working disk 5
1.10 Files in HiSoft FORTH 5

**2 Introducing FORTH 7**

2.1 The Data Stack 8
2.2 FORTH arithmetic 11
2.3 Boolean Operators 13
2.4 The Word 13
2.5 Decisions: IF ELSE THEN 15
2.6 Repetition 17
2.7 The BEGIN Loops 18
2.8 FORTH Variables and Constants 20
2.9 FORTH input/output 21
2.10 Vocabularies 22

## 3 HiSoft FORTH User Manual 25

**3.1 HiSoft FORTH Screen editor** 25
**3.2 The Terminal** 28
**3.3 The Shell** 28
3.3.1 File Menu 29
3.3.2 Edit Menu 29
3.3.3 Screen Menu 30
3.3.4 FORTH menu 30
3.3.5 Shell Restrictions 30
**3.4 The Program Compiler** 31
**3.5 FORTH as a desk accessory** 31
**3.6 The Tool Editor** 32
**3.7 The Ramdisk** 33
**3.8 The Disk System** 33
**3.9 Using ASCII Files** 35
**3.10 Converting block files to/from ASCII files** 36

## 4 HiSoft FORTH Reference 39

**4.1 Common FORTH Language Words** 39
4.1.1 Glossary Notation 39
**4.2 HiSoft FORTH specific features** 40
4.2.1 FORTH-83 standard and HiSoft FORTH 40
**4.3 FORTH-79** 61
**4.4 FORTH-83 Standard** 63
4.4.1 The Required word set 63
4.4.2 FORTH-83 standard and HiSoft FORTH 63
4.4.3 FORTH-83 Words 64
**4.5 ONLY Words** 68
**4.6 Double Number Extension Set** 70
**4.7 Floating point** 72
**4.8 Assembler** 74
4.8.1 Loading the Assembler 74
4.8.2 Using the Assembler 75
4.8.3 Assembler Syntax 75
4.8.4 Using HiSoft FORTH with DevpacST 76

4.9 Multi-Tasking 77
4.10 MIDI library 79
4.11 Graphics 80
4.12 Atari ST Extensions 81

5 Direct Operating System Calls 83

5.1 GEMDOS 83
5.2 The BIOS 88
5.3 The Extended BIOS (XBIOS) 90
5.4 GEM VDI (Virtual Device Interface) 93
5.4.1 GEM VDI arrays 93
5.4.2 GEM VDI Control Functions 94
5.4.3 GEM VDI Output Functions 98
5.4.4 GEM VDI Attribute Functions 105
5.4.5 GEM VDI Raster Operations 113
5.4.6 GEM VDI Input Functions 115
5.4.7 GEM VDI Inquire Functions 120
5.4.8 GEM VDI Escape Functions 122
5.5 GEM AES 124
5.5.1 GEM AES Application Library Routines 126
5.5.2 GEM AES Event Library Routines 129
5.5.3 GEM AES Menu Library Routines 132
5.5.4 GEM AES Object Library Routine 134
5.5.5 GEM AES Form Library Routines 137
5.5.6 GEM AES Graphics Library 139
5.5.7 GEM AES Scrap Library Routines 144
5.5.8 GEM AES File Selector Library 145
5.5.9 GEM AES Window Library Routines 145
5.5.10 GEM AES Resource Library Routines 150
5.5.11 GEM AES Shell Library Routines 151

Appendix A. Implementation details 153

A.1 Memory Map 153
A.2 HiSoft FORTH Compiler 153
A.3 HiSoft FORTH Headers 154

Appendix B FIG 155

Appendix C Technical Support 157

C.1 Technical Support 157
C.2 Upgrades 157

Bibliography 159

FORTH Books 159
ST Books 159

Index 163

# 1 Introduction

## 1.1 FORTH History

Charles H Moore began the development of FORTH in the late 1960s in order to provide himself with a programming language to use in the field of Astronomy as well as general purpose programs. Development began on an IBM 1130 and after tests on different machines and in different languages, the language was first coded in FORTRAN, then assembly language and then in FORTH itself.

The first application was in 1971 when Moore implemented a program for the acquisition of astronomical data at the National Radio Astronomy Observatory. Moore was sufficiently satisfied with his work that in 1973, with Elizabeth Rather, the first FORTH programmer, he founded FORTH Inc for the development of FORTH programs and systems.

As the use of FORTH grew the FORTH Interest Group (FIG) was formed, in San Carlos, California, with the aim of increasing knowledge of the language. They published listings in assembler for many different processors. FORTHs implemented from the listings took the name of fig-FORTH.

In 1979 the FORTH Standards Team published a Standard known as FORTH-79. The FST issued a revised Standard in 1983. The FORTH-83 Standard defined a 'Required Word Set' that is the minimum standard of most commercial FORTHs today.

## 1.2 About HiSoft FORTH

**HiSoft FORTH** is a fast 68000 based FORTH for the Atari ST computer. It supports FORTH-83 Standard, FORTH-79 Standard and fig-FORTH programs.

A full interface to GEMDOS through the BIOS, XBIOS, GEM VDI and GEM AES functions is included as FORTH words. There is also a FORTH full screen editor in the main kernel. It also has the ability to handle standard text files, so that other editors may be used if you wish.

**HiSoft FORTH** is a 32 bit FORTH with the stack and all arithmetic and numeric conversion using 32-bit numbers.

**HiSoft FORTH** is fig-FORTH compatible and will run fig-FORTH programs. It is also FORTH-83 Standard, after executing `FORTH-83`, as set down in the FORTH-83 Standard by the FORTH standards team.

To run FORTH-79 standard programs execute `79-STANDARD` and any FORTH-79 words that need changing from fig-FORTH will be searched first. The default FORTH standard is fig-FORTH which allows for most FORTH programs to be run immediately by **HiSoft FORTH**.

# 1.3 Acknowledgements

The source of much of the documentation in the Reference section of this manual is the document 'FORTH-83 STANDARD' by the FORTH Standards Team.

To the extent that text from that publication has been used the authors acknowledge the copyright of the FORTH Standards Team and their consent to reproduction. Thanks are due to Mr Nicholas Spurrier, George Chkiantz and Gil Filbey for useful suggestions about the software.

**HiSoft FORTH** was developed on an Atari 520 ST, using the **HiSoft Devpac ST** 68000 Assembler by Henry McGeough.

# 1.4 How to use this manual

Everyone should read the rest of this section as it describes what to do before using the package.

**Section 2** is designed for new-comers to FORTH, although the sub-section on vocabularies may be useful to experienced FORTH programmers.

**Section 3** contains the full details of the **HiSoft FORTH** built-in words, together will the differences between the different FORTH standards and how to use the turtle graphics, multi-tasking and machine language features of **HiSoft FORTH**.

**Section 4** describes the low-level interfaces to the ST's operating system: GEMDOS, BIOS, XBIOS, GEM VDI and GEM AES.

**Appendix A** is a summary of the implementation details of **HiSoft FORTH**.

**Appendix B** gives information on the FORTH Interest Group.

**Appendix C** describes how to obtain technical support. Please read this before contacting us.

The **Bibliography** gives details of some recommended FORTH books. For newcomers to FORTH, we would recommend 'Starting FORTH' by Leo Brodie.

# 1.5 Always make a back-up

Before using **HiSoft FORTH** you should make a back-up copy of the distribution disk and put the original away in a safe place. It is not copy-protected to allow easy back-up and to avoid inconvenience. This disk may be backed-up using the Desktop or any back-up utility. The disk is single-sided but may be used in double-sided drives.

Before hiding away your master disk make a note in the box below of the serial number written on it. You will need to quote this if you require technical support.

| Serial No: |
|---|

## 1.6 Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us after reading the licence statement. Without it you will not be entitled to technical support or upgrades. Be sure to fill in all the details, especially the serial number and version number.

# 1.7 HiSoft FORTH Disk Contents

The supplied single-sided 3.5" disk contains these files:

| | |
|---|---|
| HSFORTH.PRG | the complete version of the FORTH including the GEM shell |
| HSFORTH.RSC | the resource file for HSFORTH.PRG |
| FORTH.PRG | a version of the FORTH without the GEM shell but including the full GEM vocabulary. This is used by the program compiler as the basis for new versions. It also can be used instead of HSFORTH if you don't like using GEM menus. |
| KERNEL.PRG | The smallest version of the FORTH without the GEM shell and only a minimum GEM vocabulary. This is described in **Section 3.4** |
| README.TXT | See below for details. |
| FORTH.BLK | The default source file, containing the program compiler and floating point routines. See below. |
| ASM.SEQ | The source code to the assembler. See **Section 4.8**. |
| FLOAT.SEQ | The source code to the floating point library. See **Section 4.7.** |
| OBJECT.SEQ | Dick Pountain's well known Object Oriented extensions to FORTH. See the Bibliography for more sources of further information. |
| UBIK.SEQ | A Rubik's Cube demonstration program. See **Section 3.3.5**. |
| BOX.SEQ | A colour box drawing program. |
| HF8K.ACC | Desk accessory version of HiSoft FORTH. See **Section 3.5.** |
| PROGRAM.SEQ | The main program compiler for producing standalone code. See **Section 3.4.** |
| PRGINIT.SEQ | Subsidiary files used by the program compiler |
| PRG.SEQ | |
| BLK.SEQ | Contains words for converting BLK files to SEQ files and vice versa. See **Sections 3.9** and **3.10.** |
| MIDI.SEQ | A MIDI example file. See **Section 4.10.** |
| PRIMES.SEQ | A FORTH version of the famous Sieve of Erastothenes benchmark. |

## 1.8 The README File

As with all HiSoft products **HiSoft FORTH** is continually being improved and the latest details that cannot be included in this manual may be found in the README.TXT file on the disk. This file, if present, should be read at this point, by double-clicking on its icon from the Desktop and then clicking on the Show button. You can print it by clicking on the Print button.

The README.TXT file may also be read by any standard text editor.

## 1.9 Making a working disk

From your back up disk copy these files to a new blank disk:

```
HSFORTH.PRG   HSFORTH.RSC   FORTH.BLK
```

If you wish to use the stand alone program compiler then copy

```
FORTH.PRG        PROGRAM.SEQ      PRGINIT.SEQ      PRG.SEQ
```

as well.

If you wish to use the assembler or other extensions immediately, then copy the appropriate `.SEQ` file (`ASM.SEQ` for example).

Now load HiSoft FORTH by double-clicking on `HSFORTH.PRG`.

We will store our FORTH programs in the file `FORTH.BLK`. Initially this just contains a few screens for re-compiling parts of the system. We need not be concerned with these yet, but we will need some more space for our own programs. So type

```
96 MORE
```

This will give us an extra 96k in the screen file, FORTH.BLK. Newcomers to FORTH should now continue with the quick tour in the next section.

## 1.10 Files in HiSoft FORTH

HiSoft FORTH provides two methods of storing source code: traditional FORTH screen-based (or block-based) files and sequential (or ASCII) files.

Screen-based files usually have an extension of `.BLK`; these are the files that the built-in FORTH editor uses and are great for small programs and whilst learning the language. For the technically minded, .BLK files are organised into screens containing 16 lines of exactly 64 characters; there are no end of line characters but lines shorter that 64 characters are padded with spaces.

Sequential files have exactly the same format as standard ASCII text files and so you may edit these with any standard text editor such as **Tempus 2** or the editors that are supplied with **HiSoft BASIC** and **DevpacST**. They conventionally have an extension of `.SEQ`. Although such files may not be edited with the FORTH editor, they have the advantage that they require much less disk space than `.BLK` files. As such, we have used `.SEQ` files for most of the source code supplied with **HiSoft FORTH**.

You do not need to make a firm choice between the two methods of storing files, because you may convert between them. This is described in detail in **Section 3.10.**

# 2 Introducing FORTH

This section is intended to provide the FORTH novice with an insight into using the language; it is not meant to be a full tutorial. To this end many of the finer points of the language have been left out and it is suggested that the Bibliography be consulted for further reading on the subject.

This section assumes that you have successfully made a working disk as described in **Section 1.9** and that you have some knowledge of using the BASIC language.

Okay, so what is FORTH and what makes it so different from other languages used on today's computers? A simple way of describing it is to compare it with the English language. If we examine an English dictionary we can see that it consists of many words, each word usually having a specific interpretation or meaning.

These words can be combined (in a certain order) to produce many different types of sentence which again convey a meaning of their own. In turn, we can combine sentences to form paragraphs, which ultimately can be made into a complete novel if we so require. The point we are driving at here is that from relatively few words one can produce an infinitely extensible language.

FORTH to some extent is like English. It too has a *dictionary* which is composed of *words* (yes, that is the correct name). These *words* are in fact subroutines which when executed perform a specific task. *Words* may be grouped together to form new *words*, the new *word* when executed combining all of the functions of the *words* making up its definition. This process can be repeated until you finish up with one *word* which when executed runs a complete program.

One of the major beauties of FORTH is that these *words* can be executed and tested as soon as they have been created (try doing that with BASIC, Pascal or C). Another good point of FORTH is that as we create new words which are placed in our *dictionary* they become part of the language and not an independent function. For this reason it is perfectly reasonable to extend the language by writing new control structures or even a whole new compiler.

Let us make a start by examining the FORTH dictionary. With the system up and running type in the FORTH word `WORDS` and press `Return`.

 MAKE SURE THAT YOU TYPE THE NAME IN UPPER-CASE ONLY. YOU WILL GET AN ERROR MESSAGE IF YOU USE LOWER CASE IN THIS INSTANCE. This is why HiSoft FORTH will automatically set CAPS LOCK on when it loads. In fact there is a way to make FORTH ignore the casing of letters, but as upper case is traditionally used, we'll ignore this for now.

You should see the screen fill with an abundance of apparently-foreign words. These words are the subroutine names we have already spoke of and if you look carefully you will see that the word `WORDS` is among them. When we typed this in and pressed `Return` the subroutine corresponding to `WORDS` was executed. The result of this, as you have probably guessed, was to list the dictionary (or *vocabulary* ) to the screen.

What exactly happened was that after pressing `Return` a part of the FORTH system known as the *keyboard interpreter* examined the characters you typed at the keyboard and initially assumed that they formed a valid dictionary word. It then searched through the dictionary to locate the word and execute the subroutine associated with it. If it cannot find the word the keyboard interpreter then checks to see if a number was entered and acts accordingly (more of this later). If this fails then the interpreter gives up and issues an error message saying that the word was not defined in the dictionary. You can try this by typing in some random characters at the keyboard and pressing `Return`.

We can now go one step further by typing in several words on one line, for example:

```
WORDS WORDS WORDS
```

and press `Return`. The dictionary will now be listed to the screen three times in succession. If you now have a quick look at **Sections 3** and **4** you will see all of the dictionary words listed and explained in detail. It is a good idea to read through these formal explanations as you meet them in this section but don't worry if they don't make a lot of sense at first; they will with practice!

Before we move on to something more ambitious it's worth noting that all of the words in the dictionary have names made up of *UPPER CASE* characters and as such must be referenced in the same way. If you try typing, for instance,

```
words
```

then an error message will be printed on the screen to the effect that FORTH has not been able to find the word in the dictionary. When you come to create your own word definitions you are free to use upper- or lower case-characters as you so desire.

## 2.1 The Data Stack

FORTH is known as a *stack-orientated language.* This means that all variables and constants (i.e. numbers) are passed to the various FORTH words via an area of memory known as the *DATA* or *PARAMETER STACK.* This can be likened to an office 'in tray' where new items are put on to the top of the pile of existing papers, and as it is easier to look at the top item rather than search through the pile, they are usually removed in that order. For this reason the stack is usually referred to as a *LIFO* or *last in first out* stack. The last item placed on the stack is known as the *top of stack* or `TOS` with the next item down being known as `NOS` (*Next item On Stack*) and so on. Let's illustrate this with an example.

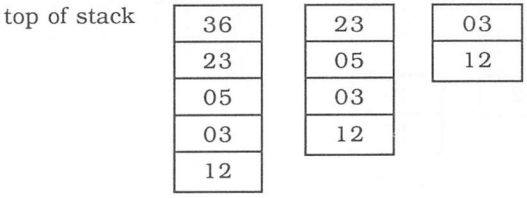
Type in the following (we will assume by now that you know that you have to press the `Return` when finished):

```
12 3 5 23 36
```

Note that the numbers are separated by at lest one space, as a space acts as a delimeter or separator between FORTH words.

What we have done here is to type in five numbers which the keyboard interpreter recognises as such. The result is that these numbers are placed on the data stack in the order that they have been typed in, with the number 36 as the last item entered and therefore with this number as the top of stack.

If we now type in the FORTH word `.` (pronounced *dot*) we will see the top stack entry (which is 36) displayed on the screen. Repeat this twice more and we will see the numbers 23 and 5 displayed. If we now continue to type . (dot) we will see the remaining numbers displayed until we get an error message printed on the screen. This message informs us that the stack is empty and that we have tried to remove a non-existent number from the data stack. What the FORTH word . (dot) actually does is to remove a number from the top of the data stack and display it on the current output device, which in this case is the screen. The diagram below shows the stack in various stages of removing the numbers.

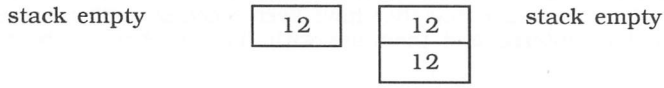



| | | | | |
|---|---|---|---|---|
| Typing: | . | . . | . . . | |
| Displays: | 36 | 23 5 | 3 12 | error |
| | (a) | (b) | (c) | (d) |

The data stack in FORTH is of fundamental importance to the philosophy of the language. As already mentioned, any FORTH words that may require variables or constants to work on obtain these from the stack. It is up to the programmer to ensure that the correct values and quantity are placed on this stack prior to a FORTH word being executed or else unpredictable results are likely to occur.

Due to the stack's importance FORTH provides many built-in words which can manipulate the stack contents. For instance, we may wish to place a number on the stack and use it twice in succession. We could if we wish place the number on the stack twice; e.g. typing `12 12.` However this is boring and would at the very least send a FORTH guru into palpitations! What we should do is to use the FORTH word `DUP` (DUPlicate). This takes a number from the top of the stack and duplicates it, placing the original and a copy back onto the stack as the top and second item.

This is illustrated below:

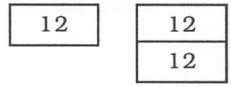



Typing: `12 DUP ..`

Displays `12 12`

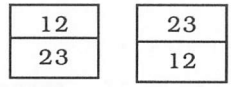
There are many more words which act upon the data stack contents. The main ones are described below in pictorial form with each diagram including a brief description of the function. We will see how these words become useful in program-writing very shortly.

i) `DUP` (Copies the top stack entry)



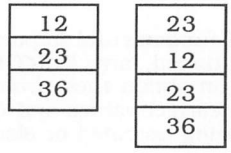

before after

ii) `SWAP` (Swaps the top two stack entries)



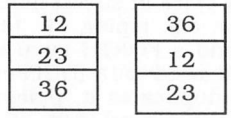

before after

iii) `OVER` (The second stack entry is copied to TOS)



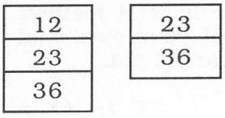

before after

iv) `ROT` (The third stack entry is removed to TOS)



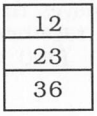

before after

v) `DROP` (Discards the top stack entry)




before after

vi) `SP!` (Clears out the stack of all entries)




stack empty

before after

Before we move on to the next section spend a little time putting numbers on the stack and then trying the words given above to see the effects on the stack contents. Remember that to put a number on the stack, you need only to type it in. To display it, use the FORTH word . (dot).

## 2.2 FORTH arithmetic

FORTH is a somewhat unusual language in the way it handles numbers and performs mathematical operations on them. Initially HiSoft FORTH deals with 32-bit integer numbers, resulting in a range of -2147483648 to +2147483647. Thus for most integer calculations, we don't need to worry about arithmetic overflow unlike implementations that use 16 bit integers. HiSoft FORTH also has facilities for even larger integers and floating point numbers: however we recommend that you learn 'ordinary' FORTH before using these extra words described in **Sections 4.6** and **4.7**.

The second and perhaps most difficult-to-understand peculiarity of FORTH number-handling is the way we perform arithmetic. For example, in the BASIC language if we wished to add two numbers together and print the result we could write:

```
PRINT 12 + 48
```

which would print the result `60` on the screen. To achieve the same result in FORTH we must type in the following:

```
12 48 + .
```

To add together (or perform any other mathematical function) in FORTH we must place the numbers on the data stack **before** we carry out the mathematical operation. This is known as *post fix* or *reverse polish* notation and was used on early electronic calculators. In the example above we place two numbers 12 and 48 on the stack and then we use the FORTH word + (plus), which removes the top and second stack entry, adds them together and places the result of the addition back on to the top of the stack. This result is then displayed using . (dot).

Let's try some further and slightly more complicated examples to gain a little familiarity with post fix notation. We will show the normal method of writing the expression followed by the FORTH method. Make sure you try these examples and perhaps a few of your own, and try to visualise the stack contents as the operations are carried out.

a) `12 + 34 + 45 = 91` `12 34 45 + + .`

Notice here that we have to use two + signs. The first stage of the addition by FORTH is to take the top two stack items and add them together; i.e. `45` and `34` placing the result `79` on the top of the stack. The second plus sign is then executed by FORTH which again removes the top stack item (now `79`) and the second stack item `12`, adds them together and puts the final result `91` on the stack where it is printed with . (dot) which removes teh result from the stack.

b) `45 - 12 = 33` `45 12 -`

c) `68 - 3 + 12 = 77` `68 3 - 12 +`

or `12 68 3 - + .`

In this example, where we are using mixed operators, we must ensure that we place either the numbers or the operators in the correct order. As you can see from above there is more than one way to achieve the same result. We can either place the first two numbers `68` and `3` onto the stack and subtract them followed by the third number and addition sign. Alternatively, we could place all three numbers onto the stack in one go and then compute the expression by first subtracting the top two stack numbers (`68 - 3`) and then adding `12` to the result.

d) `12 * 6 = 72` `12 6 * .`

e) `15 / 3 = 5` `15 3 / .`

These last two examples use the multiply and divide operators of FORTH. The main point to remember here is that when using the division operator, the second stack entry NOS is divided by the top stack entry TOS. Also recall that FORTH is integer-based so if you try to divide a number which would normally leave a fractional part within the result this fractional part will not be calculated.

These then are the basic mathematical operators that FORTH provides. The list is by no means complete but it is impossible to deal with them all in this short introduction to the language. FORTH also provides certain words which act upon double-length numbers.

As a final example which incorporates the mathematical and stack manipulation words examine the expression below and the FORTH solution. Note how we use the FORTH word `SWAP` to rearrange the stack, making our calculation easier to perform.

$\frac{(10-4)}{2} = 3$ `2 10 4 - SWAP / .`

# 2.3 Boolean Operators

*Boolean operators* are words in FORTH which compare two stack values in various ways and place a -1 or 0 on the stack as the result. For example, we may wish to compare two numbers to see if they are equal in value. To do this we could write :

```
10 20 =  (TOS would be 0 as the numbers are not equal)
```

or

```
12 12 =  (TOS would be -1 as the numbers are equal)
```

The numbers 0 and -1 are also called FALSE and TRUE as they reflect the identity of the result. In the first example the numbers were not equal therefore the result of the test was FALSE. In the second example they were equal so the result was TRUE. We could have tested for the numbers being not equal to each other using the FORTH word <>. This would return a -1 or TRUE if the numbers were not equal and a 0 or FALSE if the numbers were equal.

Other Boolean operators available in FORTH are listed below with short examples to show their operation.

> (Greater than)

```
16 15 >  (TOS is TRUE as 16 is greater than 15)

15 18 >  (TOS is FALSE as 15 is less than 18)
```

< (Less than)

```
16 15 <  (TOS is FALSE as 16 is greater than 15)

15 18 <  (TOS is TRUE as 15 is less than 18)
```

0= (Equals zero)

```
12 0=  (TOS is false as 12 is not equal to zero)

0 0=  (TOS is TRUE as 0 is equal to zero)
```

As with the stack manipulators we have shown only a selection of the Boolean operators. We will meet these again shortly when we will see their importance in decision-making and controlling the flow of a FORTH program.

# 2.4 The Word

We have already stated that a FORTH word is in fact the name of a particular subroutine which is executed as soon as the word is typed in and Return is pressed. We have also said that we can string words together to execute a series of subroutines. Armed with this knowledge it is now time to write our first FORTH program.

Type in the following, taking care with spaces etc. :

```
: GREETINGS CR ." Hello world" CR ;
```

When you have typed this in and pressed Return you should be presented with the by-now familiar ok prompt. If you get an error message instead, don't panic, just type in the FORTH word COLD and try again (and watch your spelling and spaces this time). The word CR (Carriage Return) sends to the output device a new-line control code thereby forcing any subsequent printing to start at the beginning of a new line.

Now type in GREETINGS and press Return. The screen should clear and the words Hello world should be displayed with the ok prompt at the beginning of the next line. If we now list the dictionary by typing WORDS we should see the word GREETINGS at the top.

We have in fact created a new FORTH word or definition named GREETINGS. Every time this word is now typed in the Hello world greeting will be displayed. To understand fully what has happened we must look more closely at the action of the keyboard interpreter. As we have already mentioned, the keyboard interpreter scans the input line interpreting (if possible) the text it finds there. In this case the first thing it meets is the : (colon). This is just another FORTH word and in fact executes a routine which switches from the command mode we are working in to the compile mode.

Now the next text word that the keyboard interpreter finds is assumed to be the name of the new FORTH word we are creating, which in this case is GREETINGS. This word is then created in the dictionary as the latest entry. The words which follow the name now have their runtime addresses compiled into the new definition to form the executable part of the word. In our example above the word CR is the equivalent of a BASIC PRINT statement with nothing following it: it causes the cursor to advance to the next line. The FORTH words ." (dot quote) and " (quote) are analogous to BASIC's PRINTing of a string. Anything that appears between ." and " will be printed on the current output device.

Note that there must be a space between the ." and the first character of the text string we are printing. The word CR prints a carriage return on the output device while the ; (semi-colon) ends compilation and returns to the command or immediate mode.

Another simple but more useful example to illustrate the creating of new definitions is given below. This calculates the square of a number which is the top item on the stack.

```
: SQUARE DUP * ;
```

This word called SQUARE duplicates the top stack entry with DUP and then multiplies these two numbers, leaving the square of the number as the result on the stack. Try this with various numbers using . (dot) to display the result.

# 2.5 Decisions: IF ELSE THEN

FORTH, like many languages, supports the IF statement which allows program flow to be controlled depending upon the results of some previous action. The format of this in FORTH is

| | |
|---|---|
| IF | the value on the top of the stack is TRUE<br>execute the words that follow |
| ELSE | if the top stack entry was FALSE<br>execute the following words |
| THEN | Terminates the structure |

The example below illustrates the use of the IF statement in controlling program flow. Let's say we have an electronic circuit which monitors the pressure of steam in a pipe connected to a boiler. If the pressure exceeds the safety limit of 100 psi. then an alarm must be activated and a warning message displayed on a VDU screen. If the pressure is normal then we display the psi. We will assume that we have created a FORTH word called ALARM which will ring an external warning bell.

This will be rather more complicated than the previous examples, so it will be best to enter this program with the screen editor. Each FORTH file is divided into blocks. We will enter our program into block 10, by using:

```
10 ED
```

This will bring up the FORTH screen editor display. If you haven't used the editor yet, don't worry: just press the Ins key and it will behave like a normal screen editor: pressing the cursor keys will move the cursor, Backspace will delete the previous character, Del will delete the next character and Return will insert a new line.

Also, you will be able to lay out the program in a more structured and readable way as shown below.

```
: ALARM 7 EMIT ;          ( Makes a beep --- )

: TEST_PRESSURE           ( pressure --- )
DUP 100 >
IF ALARM CR ." WARNING PRESSURE OVER LIMIT" DROP
ELSE CR ." Pressure is " . ." psi"
THEN ;
```

We first of all define the word ALARM which uses an existing FORTH word EMIT to produce the necessary beep from the loudspeaker (character 7 is the bell). The main definition is called TEST_PRESSURE and requires the pressure value on the stack as its input value. Notice the brackets after the name. These are used to enclose comments in FORTH in much the same way we would use REM statements in BASIC. In this case we are indicating that the pressure value must be on the stack before execution of the definition. Any values that are returned would be indicated after the ---.The first step is to duplicate the pressure value with DUP as we will need two copies of the value for this routine. We then use a Boolean operator > to see if the pressure is greater than the maximum 100.

Note that this test removes the first pressure value from the stack, hence the reason for the DUP. If this pressure value is greater than 100 (the maximum) then a TRUE flag is placed on the stack ready for interception by the IF. If the flag is TRUE we execute ALARM, which beeps our loudspeaker and then prints the warning message on the screen. We finally DROP the remaining pressure value that is left on the stack and the program falls through, ignoring the ELSE statement, to the THEN which is the exit from the routine.

If the pressure was within limits the > test would leave a FALSE flag on the stack which would result in the IF statement not being executed. The program would fall through to the ELSE part which would print the pressure on the screen. Notice the . (dot) which takes the remaining pressure value from the stack and displays it between the two messages. The program exits via THEN.

To try out these words we will need to compile them. To do this first leave the editor by pressing F10 and then type

```
10 LOAD
```

This causes the compiler to start loading the code from block 10. If an error message is displayed then just type

```
10 ED
```

to return to the editor and re-check the program. Now when you type 10 LOAD to the interpreter, the messages

```
ALARM ISN'T UNIQUE TEST_PRESSURE ISN'T UNIQUE
```

will appear; this is quite normal, the system is just telling us that our old definitions have been superseded.

Once we have successfully compiled ALARM and TEST_PRESSURE we can try them out. Using

```
ALARM
```

on its own will cause the usual beep (assuming the volume isn't turned down!).

```
10 TEST_PRESSURE
```

will execute our new word with an argument of 10.

Try out this routine with various numbers on the stack and check to see that it works correctly. Then when you are sure it does what it is supposed to do, try and map out on a piece of paper the states of the stack as the routine goes through its various stages. This will be a very useful exercise in helping to understand the action of the IF statement and the data stack.

# 2.6 Repetition

FORTH provides several ways for repeating a series of instructions either with or without conditions. The first of these that we will look at is the DO loop. This is very similar to the FOR-NEXT loop that is used in BASIC and takes the general form shown below:

```
: LOOPTEST1  100  0  DO  I  .  LOOP  ;
```

This routine simply prints the numbers 0 to 99 on the screen. To try this out you could add this line to screen 10 and uses the same commands as before.

The upper and lower limits of the loop are set by the two numbers placed on the stack (in this case 0 and 100) with the upper limit being one higher than the actual number of times we wish the loop to execute. The loop starts with a value of 0 and executes everything between the words DO and LOOP, incrementing the loop index at every pass. When the loop index reaches the limit the loop terminates without making a final pass of the loop body (this is why numbers are only printed out up to 99). A new word I has been introduced, which takes the current loop index and places it onto the stack, where we can print it with . (dot). The loop can be made to count in increments greater than one by using the FORTH word +LOOP i.e.

```
: LOOPTEST2  13  2   DO   I   .   2   +LOOP   ;
```

This will cause the loop to increment in steps of 2 beginning with 2 and finishing with 12. The loop fails at this point as the next increment after 12 would be 14 which is greater than the loop limit of 13. We can also make the loop count backwards if we require by making the loop index negative as shown below:

```
: LOOPTEST3  1  11  DO  I  .  -1  +LOOP  ;
```

Notice here that the stack limits have been reversed with the higher number, which is now the starting value of the loop, being on the top of the stack. The loop index is decremented by 1 on each pass until the lower limit is reached.

Finally, we give what may appear at first glance a rather complex example of nesting loops (placing one loop inside another). This example prints the multiplication tables from 1 to 10 on the screen in a neat format. It is a good idea to use another Again pay attention to the general layout of the routine as it shows a way of writing FORTH programs in readable fashion. It is probably a good idea to use a new screen (say number 11) for this program.

```
: TABLES                (Prints multiplication tables )
." Number"  10 SPACES  ." Multiples"  CR ( Print heading )
8 SPACES
9 1 DO  ." X" I  .  LOOP  CR
9 1 DO  2 SPACES  I  .  4 SPACES
        9 1 DO I J * . LOOP CR
LOOP ;
```

The routine begins by printing an appropriate heading for the tables. We then use a DO loop to print out the number tables. The actual tables are printed by using two loops, one nested inside the other. The outer loop sets the initial table number while the inner value calculates the actual multiplication table for that number.

There are several FORTH words introduced in this example that we have not met before. The first, SPACES, prints onto the output device the number of spaces indicated by the top stack entry. It is mainly used in formatting text as in the example here.

We have already met the word I which takes the current loop index and places it onto the stack.When we are nesting loops, the word J takes the index of the outer loop (if the loops are nested) and places this on the stack. There is also a further word K which is not used in this example but places the index of the outer loop in a three-level nested DO loop onto the stack.

There is one more FORTH word relating to the DO loop that is of interest and that is the word LEAVE. This word is usually used in conjunction with an IF statement and when encountered forces the loop to terminate immediately by setting the loop index to its maximum value.

## 2.7 The BEGIN Loops

FORTH provides several ways of repeating a sequence of instructions based on the word BEGIN. The first of these is the BEGIN-AGAIN loop and is illustrated in the example below.

```
: INFINITE BEGIN ." This goes on and on and on" AGAIN ;
```

As can be seen from the example this sets up an infinite loop that cannot be exited. Its main use is in the main body of a program where the program is usually of a closed-loop nature. Be warned!! If you type in this example and execute it you will be able to exit only by resetting the computer and reloading FORTH.

The second BEGIN loop is the BEGIN-UNTIL and takes the general form as shown below.

BEGIN — This marks the beginning of the loop.

Any sequence of FORTH words which form the main body of the loop. This body should result in the stack holding a TRUE or FALSE flag which is used by the next phase.

UNTIL — This word removes the flag from the stack and if it is FALSE returns to the code after BEGIN thereby executing the loop once again. If the flag is TRUE the loop is exited and control continues after the UNTIL.

On the next page, there's an example: once again it is probably best to enter this on a new screen.

```
: YES_NO ( --- 1=yes 0=no )
BEGIN
." Please answer Yes or No Y/N"
KEY 32 OR DUP 121 =
IF -1 1
ELSE DUP 110 =
IF 0 1 ELSE CR ." PLEASE ANSWER YES OR NO (Y/N)" DROP 0
THEN
THEN
UNTIL
SWAP DROP ;
```

This example reads a character terminated by Return from the keyboard and tests for a 'yes' or 'no' response. If the response is Y or y then a TRUE flag is placed on the stack. If the response is N or n then a FALSE flag is placed on the stack. An invalid keyboard entry will cause an error message to be printed and the loop repeats.

The loop itself begins by printing the keyboard prompt message. The FORTH word KEY reads a single character from the keyboard and places the ASCII value of this key on the stack. We then perform a neat little bit of logical manipulation by first removing the stack value and bitwise-ORing it with decimal 32 to convert it to a lower-case character (the result is placed back on the stack). If the character is already upper-case ASCII then this ORing makes no difference.

We then DUPlicate the stack entry for later use and test to see IF it is equal to decimal 121 (which is the ASCII code for lower-case y. If this test is true then we place the numbers 1 and 1 on the stack and the routine drops through to the UNTIL where the top stack item is removed (a 1) and is tested by the UNTIL. As this value is TRUE the routine is exited leaving the last 1 on the stack which indicates a 'yes' response.

If the test for a y fails we then test for a N response by DUPlicating the stack value once more and testing for equality with decimal 110 (lower-case n). If this is correct we place the numbers 0 and 1 on the stack and fall through to the UNTIL where the 1 is removed from the stack and the loop exited leaving a 0 (indicating a 'no' response). If the 'no' test fails we can assume that an invalid key has been pressed. Control will transfer to the second ELSE statement which then prints an error message, drops the inputted stack data and places a 0 on the stack. When the UNTIL takes this value from the stack it will cause the program to return to the keyboard entry prompt.

One important point that this program illustrates is in controlling the stack. It is the programmer's responsibility to ensure that the stack is set up correctly for parameter-passing to FORTH words and just as important, that the stack is cleared of all unused values. The stack itself is not infinite in length and bad stack handling (especially in a loop) can soon fill the stack, causing the program to crash without warning. If at any time you are not sure of the contents of the stack but wish to clear it, you can use the FORTH word SP! which clears all values from the stack.

The final BEGIN structure that FORTH uses is the BEGIN-WHILE-REPEAT loop.

BEGIN Marks the beginning of the loop.

Any sequence of FORTH words which result in a Boolean flag on the stack.

WHILE The word WHILE which tests the stack value and if TRUE executes the next sequence of FORTH words. If the stack value is FALSE then the loop is terminated.

Any sequence of FORTH words which form the main body of the loop.

REPEAT The word REPEAT returns execution if the main body was executed to the code after BEGIN.

```
: SPACEBAR (Tests for space bar )
BEGIN
KEY 32 =
WHILE
." space bar pressed." CR
REPEAT
." space bar not pressed . Loop exited" ;
```

SPACE_BAR waits for a keyboard input and tests for the stack value being equal to ASCII 32 (the space). While this is true the response message is printed. If we type any other character the loop exits and the exit message is printed.

## 2.8 FORTH Variables and Constants

FORTH provides means for predefining variables and constants. A constant (i.e. a value which will not change once assigned) is declared in FORTH by :-

```
12 CONSTANT DOZEN
```

The FORTH word CONSTANT assigns the value 12 to the constant name DOZEN. This name is added to the dictionary in the same way as a compiled definition. Typing DOZEN will return the value 12 to the top of the stack.

A variable is defined in a similar fashion e.g.

```
70 VARIABLE TEMPERATURE
```

This time if we type TEMPERATURE we do not get the contents of TEMPERATURE but the address of the contents. If we wish to place the value of TEMPERATURE on the stack we have to use the FORTH word @ e.g.

```
TEMPERATURE @
```

and then display it with . (dot). We can if we wish combine both of these functions in one go with the word ? e.g.

```
TEMPERATURE ?
```

will display the value of TEMPERATURE on the screen.

These words @ and ? do not have to be used with constants or variables only, they will work with any memory location e.g.

```
HEX 100 ?
```

will display the contents of memory location hexadecimal 100. If you try this type DECIMAL to return to normal decimal number base.

We can modify the contents of variables or memory locations using the word ! (pronounced store) e.g.

```
50 TEMPERATURE !
```

A variation on this word is +! (plus store) which will add a number to the contents of a variable or memory location e.g.

```
12 TEMPERATURE +!      or   -12 TEMPERATURE +!
```

will add or subtract respectively 12 to the previous value of TEMPERATURE.

## 2.9 FORTH input/output

To date we have only dealt with a minor portion of input/output using FORTH namely the words CR .(dot), KEY and ." for printing a string. FORTH has many other commands which provide a very flexible means of providing input/output which can be configured to practically any I/O device. We shall begin by examining the most primitive of the output operators, the word EMIT. EMIT takes a number from the stack which represents an ASCII value and prints it at the current cursor location on the screen. For example,

```
70 EMIT
```

Will print the letter F on the screen.

We could string several of these together to print a full word as in:

```
70 EMIT 79 EMIT 82 EMIT 84 EMIT 72 EMIT
```

The above will print FORTH but this is inefficient and can be performed more easily using ." FORTH". EMIT is very useful for printing control characters or special graphic codes. A more useful FORTH word is TYPE.This requires an address and a byte count to be placed on the stack prior to its call. TYPE then prints the count of ASCII characters starting from the address onto the screen. We can illustrate the use of type by introducing an input operator EXPECT.

EXPECT is the opposite of TYPE and reads keyboard input into the address and up to the byte count specified on the stack. For example:

```
PAD 32 EXPECT
```

will read characters from the keyboard until Return is pressed or the character count of 32 is exceeded. The characters are stored (in this case) in an area of memory called the PAD , which is a scratch-pad area available to the user. Now typing

```
PAD 32 TYPE
```

will print out 32 characters stored at the address of PAD.

## 2.10 Vocabularies

**HiSoft FORTH** uses a method of vocabularies proposed by William F. Ragsdale in the FORTH-83 standard document. It uses a closed vocabulary system, with all vocabularies compiled into the an ONLY vocabulary. There are two search orders CURRENT and CONTEXT.

CURRENT is the vocabulary into which new FORTH words are compiled

CONTEXT vocabularies are the vocabularies that will be searched by FORTH when compiling new words. There are 8 slots for different vocabularies plus a place for the ONLY vocabulary. The search order is specified at run-time rather than the time a new vocabulary is created. The search order can be displayed with the word ORDER. Vocabularies are not immediate as in early FORTHs.

| | |
|---|---|
| WORDS | Displays list of words. |
| ALSO | Adds vocabularies to search order. |
| PREVIOUS | Removes a vocabulary from search order |
| ORDER | Displays search order. |
| DEFINITIONS | Sets up vocabulary for compilation. |
| VOCS | Displays list of vocabularies. |

The following vocabularies are already defined:

| | |
|---|---|
| ONLY | Root vocabulary |
| FORTH | Main FORTH words |
| FORTH-83 | FORTH-83 standard |
| 79-STANDARD | FORTH-79 standard |
| EDITOR | Words used by the editor |
| TOS | Atari TOS words inc GEMDOS, BIOS and XBIOS. |
| GEM | Atari GEM words (the AES and VDI) |
| TASK | Multi-tasking words |
| GRAPHIC | Turtle graphics |
| SHELL | The Shell. See SHELL.SEQ |

These are described in detail later in this manual.

This completes our short introduction to the world of FORTH. We have in reality barely scratched the surface of this fast and very flexible language. It is hoped that in reading this section, your appetite has been whetted to progress further by exploring the language in greater detail. The **Bibliography** gives details of a selection of books that are available on the subject.

# 3 HiSoft FORTH User Manual

## 3.1 HiSoft FORTH Screen editor

**HiSoft FORTH** comes complete with a full screen editor, resident in memory. To edit a screen simply type the screen number followed the FORTH word `ED`.

e.g. `100 ED`

would edit screen 100.

Be careful of swapping disks while in the editor, as there are 8 disk buffers. If you are in the editor and you want to swap to another disk, press the `Help` key to go into FORTH and type `FLUSH`. This will save any updated screens to disk and un-assign the disk buffers; then you can change disks.

### Editor words

`ED ( n --- )`

Edit screen `n` in 80 column high or medium resolution.

`WHERE ( blk count --- )`

When FORTH aborts loading a screen it saves the `blk` number and the offset `count` into the block. If you then type in `WHERE` it will bring up the editor with the cursor on the place where the FORTH compiler thinks there is an error.

`LOCATE ( --- )`

Used in the form

```
LOCATE <forth_word>
```

Finds the source of a FORTH word that has been compiled from a screen.

`DATE ( --- )`

Stamp the current date and 3 character name abbreviation in the top right corner of current edit screen. The name can be changed with the following code:

```
" JOE" STAMP 1+ 3 CMOVE
```

To display the current stamp use:

```
EDITOR STAMP FORTH COUNT TYPE
```

Perssing Shift F9 in editor will cause the current date to be displayed at the top right of the current block.

## Editor Keys

Once you are in the editor, you can use the cursor keys to move around the screen. Other keys are as follows:

| | |
|---|---|
| Return | Move to start of a line or, if you are at the start of a line, move to the next line. When in insert mode this will split the current line. |
| Tab | Tab forward 4 spaces. |
| Clr Home | 'Wipe' the current screen. This clears the entire screen to spaces. |
| Insert | Toggle insert mode. |
| Backspace | Move cursor back and delete by one character. |
| Delete | Delete the next character and 'pull back' the rest of the line. |
| Undo | Abandon current edit screen. |
| Help | Execute one line of FORTH, then return to editor. This is a useful function for getting key values or maths or hex conversion. You can even use it to run your own FORTH editor macro programs from within the editor. |

A tip: if you try and edit a machine code screen and the screen fills with garbage,then move to another screen and use the Help key to enter a PAGE command; the editor will then redraw the screen.

## Function Keys

The function keys are used for the following:

| | | |
|---|---|---|
| F1 | Insert line. | If a line is pushed off the bottom of the screen it is saved in a 1k buffer. A whole screen could be saved in this buffer. |
| F2 | Delete line. | If there is a line in the save buffer then F2 will pull it back onto the screen at line 15. |
| F3 | Push line to Edit line buffer. | The Edit line buffer is a 1k buffer that can be used to save and restore lines from a block. It is separate from the buffer used by the insert and delete line keys. |
| F4 | Pull line from Edit line buffer. | Pulls the top line from the Edit line buffer to a block at the cursor position. |

| | | |
|---|---|---|
| F5 | Swap the top two lines in Edit line buffer. | Only the top line is displayed. |
| F6 | Copy FORTH screens. | This key can be used to copy FORTH screens. It takes 3 values; source, destination and number of screens to copy. It will copy overlapping ranges, so it can be used to shuffle the screens forward for an extra screen, if you find you need one. If you don't want to copy any screens or if you press the key by mistake, press any key to abort, as the copy function checks for the correct number of input values. |
| F7 | Next screen. | Scroll forward to the next screen. |
| F8 | Prev Screen. | Scroll backward to the previous screen. |
| F9 | Goto screen. | Go to the screen number requested. |
| F10 | Exit Editor. | |

There are also a series of commands that can be accessed by holding down one of the Shift keys and then pressing a function key, as follows:

| | |
|---|---|
| ShiftF1 | 'Pull' current edit line from the stack |
| ShiftF2 | 'Push' current edit line on to the stack |
| ShiftF3 | .'Push' current edit line on to the stack |
| ShiftF4 | 'Pull' current edit line from the stack |
| ShiftF7 | 'Grab' Block. Pulls the entire block from the line buffer |
| ShiftF8 | 'Put' Block. Pushes the entire block to the line buffer |
| ShiftF9 | Inserts the system date at the bottom right of screen |
| ShiftF10 | Exit editor without saving. |

By holding down the Shift key as you press the functions keys F2-F3 the current edit line will be pushed to the top of the line stack.

By holding down the Shift key as you press the functions keys F1 or F4 the current edit line will be pulled from the line stack.

## CTRL- Keys

The editor uses the following control keys:

| | |
|---|---|
| E | Move cursor up. |
| X | Move cursor down. |
| S | Move cursor left. |
| D | Move cursor right. |
| R | Goto previous screen. |
| C | Goto next screen. |
| Y | Delete line. |

| | |
|---|---|
| Q | Clear line. |
| V | Insert mode. |
| G | Delete character. |
| F | Goto next word. |
| Z | Copy line to buffer. |
| W | Paste line from buffer. |
| T | Trash (clear) edit screen. |
| B | (Binary). Treat the block as one 1024 character line |
| U | Split line |
| N | Join line |

## 3.2 The Terminal

**HiSoft FORTH** uses a scrolling 16 line buffer for text input. The following keys are used by the terminal:

| Key | Function |
|---|---|
| Insert | Insert a space. |
| Delete | Delete a character. |
| Backspace | Backspace over character. |
| <- | Move backwards. |
| -> | Move forwards. |
| Up arrow | Scroll edit buffer backwards. |
| Down arrow | Scroll edit buffer forwards. |
| Clr Home | Clear edit line. |
| Undo | Undo edit. |

The terminal accepts input from the start of a line up to the cursor.

## 3.3 The Shell

**HiSoft FORTH** is supplied with a GEM Shell program. This allows the use of desk accessories and the use of many FORTH operations from pull down menus. The source for the shell is supplied and, along with the FORTH kernel, you can use it to compile a new shell of your own. If you want to make new menus or dialog boxes, we recommend using a Resource editor, such as **HiSoft WERCS**.

The menus are as follows:

| | |
|---|---|
| Desk | Access desk accessories. |
| File | Use ramdisk and Quit option. |
| Edit | Some editor options. |
| Screen | Work with screens. |
| FORTH | Use different FORTH Tools. |

## 3.3.1 File Menu

| | |
|---|---|
| New | Clear the ramdisk. |
| Open... | Open file from file selector and read into ramdisk. |
| Close | Clear ramdisk and login disks. |
| Save | Save ramdisk to disk as RAMBLK. |
| Save as.. | Save ramdisk as name selected from file selector. |
| Exit to DOS | Exit to full TOS screen. |
| To Output... | Print a range of Triads. You are prompted to enter a range of screens. These screens will be printed. Some other screens may be printed. Only those knowing the secret of the Triad will know when and why! |
| Quit | Quit to Desktop. |

## 3.3.2 Edit Menu

| | |
|---|---|
| Edit... | Edit screens. |
| Undo | Undo current screen. |
| Cut | Cut line to line buffer. |
| Copy | Copy line to line buffer. |
| Paste | Paste line from line buffer. |
| Wipe | Wipe current screen.This clears the entire screen to spaces. |
| Exit | Exit editor. |

## 3.3.3 Screen Menu

| | |
|---|---|
| List | List current screen. |
| Next | List next screen. |
| Prev | List previous screen. |
| Goto... | Goto a screen. |
| Ram | Select ramdisk as default. |
| DR0 | Select DR0 (A:) as default. |
| DR1 | Select DR1 (B:) as default. |
| DR2 | Select DR2 (C:) as default. |
| Load... | Load current screen. |
| Copy... | Copy FORTH screens. |

## 3.3.4 FORTH menu

| | |
|---|---|
| Words | Display words of transient vocabulary. |
| Order | Show CONTEXT and CURRENT search order. |
| Forth | Set ONLY FORTH search order. |
| Forth-79 | Set 79-STANDARD search order. |
| Forth-83 | Set FORTH-83 search order. |
| GEM | Set ALSO GEM. |
| TOS | Set ALSO TOS. |
| Vocs | Show vocabularies. |
| Tool | Edit memory. |
| Index | Display the next 20 screen titles. |
| Dir | Display a directory listing. |
| Clr Page | Clear output screen. |
| Triad | Print current triad. i.e. print 3 FORTH screens one of which will be the current screen. |
| Print | Print screen dump. |

## 3.3.5 Shell Restrictions

Because the Shell uses a resource file, you cannot run programs under it which use their own resource files. Therefore such programs should be run under FORTH.PRG rather than HSFORTH.PRG. Alternatively you may use the Exit To DOS shell command to 'shut down' the shell.

The shell also has full GEM initialisation and uses the Line-A VT52 clipping. Thus some graphics oriented programs may not work as expected under the Shell.

For example, the Rubik's Cube program supplied in `UBIK.SEQ` should only be run from `FORTH.PRG` or after shutting down the shell. It may be run after copying `UBIK.SEQ` to the working disk suggested in **Section 1.9** using the following:

```
FLOAD  UBIK.BLK
```

It may be compiled to disk using screen 6 of the FORTH.BLK file supplied,i.e. using:

```
6  LOAD
```

## 3.4 The Program Compiler

The compiler is loaded from FORTH source. To compile a FORTH program as a stand alone `.PRG` program perform the following steps :

1. Load compiler. i.e. `FLOAD  PROGRAM.SEQ`

2. Load your program. e.g. `10  LOAD`

3. Patch `STARTUP` e.g. `'  MYAPPL  STARTUP  !` where `MYAPPL` is the word which you wish to execute when the stand alone program run.

4. Save the program. e.g. `"  MYPROG.PRG"  PRG_SAVE.` This will save the standalone version as `MYPROG.PRG`.

Now, when the application is double-clicked from the Desktop, your word `MYAPPL` will run.

Initially the compiler is set up to base the code on `FORTH.PRG`. If your application doesn't use the GEM vocabulary then `KERNEL.PRG` can be made the basis by changing the code `PROGRAM.SEQ` from `FORTH.PRG` to `KERNEL.PRG`. This will make your application about 10k smaller.

You can even use this technique to produce a version of the FORTH Desk Accessory which does not support the GEM functions.

## 3.5 FORTH as a desk accessory

The desk accessory version of FORTH that is supplied on the master disk has 8K bytes of workspace, hence the name, `HF8K.ACC`. This works just like a normal version of **HiSoft FORTH** except that it does not contain the GEM shell and you can use it from inside any GEM program.

This section describes how to make a version of **HiSoft FORTH** that has a different amount of workspace.

Make sure that the disk that you are using contains `FORTH.PRG` as well as `HSFORTH.PRG`, `HSFORTH.RSC` and `FORTH.BLK`.

Use

```
4 ED
```

to edit screen 4 and this will display the code that will produce a new desk accessory. You should change the constant 8192 to be the desired amount of workspace. It is also a good idea to change the file name! Then return to FORTH by typing F10 and then enter

```
4 LOAD
```

This will then write your new version of FORTH to disk. Naturally you will need to re-boot your machine to use the desk accessory version.

# 3.6 The Tool Editor

The Tool Editor allows the viewing and editing of memory or disk.There are 2 ways to invoke the Tool Editor:

- From the Tool menu option
- From the terminal in the vocabulary SHELL used in the form:

```
addr TOOL
```

Once in the Tool Editor you can use the cursor keys to move around and also the following keys :

| | |
|---|---|
| F7 | Move back 256 bytes in memory. |
| F8 | Move forward 256 bytes in memory. |
| F9 | Goto address in memory. |
| F10 | Exit the Tool Editor. |
| Help | Execute one line of FORTH code. |

The F9 option can accept FORTH words to form the address.

e.g. ' PAGE goes to the CFA of PAGE.

10 BLOCK loads block 16 into memory and goes to the block buffer. Note that the input is in hexadecimal.

To Edit a disk block:

1. Use DISK to edit direct disk blocks. e.g. DISK.
2. Set vocabulary to SHELL e.g. ALSO SHELL.
3. Edit disk buffer. e.g. 10 BLOCK TOOL.
4. Edit block. ( next 4 * 256 bytes).
5. Use UPDATE menu option to update and flush block to disk.

This allows you to use **HiSoft FORTH** as a disk editor, but you should only do this if you know what your doing. Also if you going to work on a disk it is always best to work on a copy.

## 3.7 The Ramdisk

**HiSoft FORTH** supports a ramdisk in FORTH free memory. The ramdisk is set up between LOMEM and HIMEM and can be moved to anywhere in memory by changing the values of these two variables.

It is best to set the difference between LOMEM and HIMEM to be multiples of 1024 bytes. Some words that are used by the ramdisk are as follows:

```
RAM ( --- )
```

Sets disk offset in Sysvar OFFSET to 2160, the base of the ramdisk.

```
RAMK ( n --- )
```

Sets ramdisk size to n kilobytes.

```
RAMCLR ( --- )
```

Clears the ramdisk.

```
?RAM ( --- n )
```

Returns the size of the ramdisk.

```
MEMORY ( --- n )
```

Returns size of Free Ram between HERE and LOMEM.

```
LOMEM ( --- addr )
```

Variable containing the lowest address of the FORTH ramdisk.

```
HIMEM ( --- addr )
```

Variable containing the highest address of the FORTH ramdisk.

```
RAMDISK ( buff_addr blk_number R/Wflag --- )
```

Ramdisk handler used by R/W.

## 3.8 The Disk System

The **HiSoft FORTH** Disk system is set up as 11 logical devices of up to 1000 blocks. The Disk system will default to the first .BLK file it finds, on the first access of the disk.

# Disk Map

| Disk | Blocks | Drive Usage |
|---|---|---|
| DR0 | 0 -> 999 | Disk 0 or Direct access A: |
| DR1 | 1000 -> 1999 | Disk 1 or Direct access B: |
| DR2 | 2000 -> 2999 | |
| DR3 | 3000 -> 3999 | |
| DR4 | 4000 -> 4999 | |
| DR5 | 5000 -> 5999 | |
| DR6 | 6000 -> 6999 | |
| DR7 | 7000 -> 7999 | |
| DR8 | 8000 -> 8999 | |
| DR9 | 9000 -> 9999 | |
| | 10000 -> | ramdisk |

Typing ONLINE from the terminal will show logged on drives.

e.g.

```
ONLINE
DR0:  FORTH.BLK
DR1:  MYPROG.BLK
DR2:
RAM:
```

DISK ( --- )

This word will set DR0 and DR1 to be accessed as direct disk blocks. (360 blocks for single-sided and 720 blocks for double-sided disks). If you use a disk this way it should only be used for FORTH.

FILE ( --- )

Reset DR0 and DR1 to file access.

As well as the defaults and direct access there are other words that control disk access.

_: ( --- )

This word will set the disk to be accessed for file access. e.g.

| | |
|---|---|
| A: | set drive 0. |
| B: | set drive 1. |
| C: | set drive 2 (hard disk). |
| D: | set drive 3 (ramdisk). |

$CD ( addr --- )

Change the current Sub-Directory. Use as follows:

```
" DOS" CD
```

Change directory to DOS.

`$OPEN ( addr --- )`

Set current drive to filename `addr`. Use as follows:

```
" MYPROG.BLK" USING
```

Sets current file stream to `MYPROG.BLK`.

`$MAPS ( addr n --- )`

Set drive n to filename `addr`. For example,

```
" MYPROG.BLK" 1 MAPS
```

Sets `DR1` stream to `MYPROG.BLK`.

`MAKEFILE ( --- )`

Create new file. Use as follows:

```
MAKEFILE  NEWFILE.BLK
```

This would create a new file called `NEWFILE.BLK`.

`MORE ( n --- )`

Adds `n` more blocks to the current file.

`LOGOUT ( --- )`

Closes all open files. This word is called by `BYE`.

## 3.9 Using ASCII Files

This section describes how to use ASCII rather than block-based files with HiSoft FORTH. Such files have the advantage that they can be read and written with just about any word processor or program editor. The recommended file extension for ASCII based FORTH files is `.SEQ` (short for sequential).

**HiSoft FORTH** will even let you invoke your favourite editor once you have loaded FORTH. Before installing an editor you will need to ensure that you have returned enough memory to the system for the editor to run. This is specified by indicating how much memory you which to keep for FORTH's workspace. For example to keep 100k for FORTH use,

```
100 1024 * SHRINK
```

This 100k does not include the size of FORTH itself.

`EDINSTALL (addr --- )`

This word is used to set up the name of the file that will be loaded as the current editor. It takes the address of a string which may include a full pathname, for example,

```
" TEMPUS.PRG" EDINSTALL
```

```
FEDIT ( <name> --- )
```

FEDIT is used to invoke a text editor from inside **HiSoft FORTH**; it should be followed by the name of the file to edit, for example,

```
FEDIT MYFILE.SEQ
```

This will invoke TEMPUS.PRG with the command line MYFILE.SEQ assuming that EDINSTALL has been used as above. Note that the file name is specified *after* FEDIT and should not be enclosed in quotes. When you exit Tempus you will be returned to FORTH. We recommend that you *not* use this from the Desk Accessory; the machine will probably crash.

```
FLOAD ( <name> --- )
```

FLOAD is used to compile an ASCII sequential file. It is the counterpart of LOAD for block-based files. So with our example file we would have

```
FLOAD MYFILE.SEQ
```

FLOAD can also be used within an FLOADed file, or you can useINCLUDE instead..

```
$FLOAD ( addr --- )
```

$FLOAD is a version of FLOAD that takes an address on the stack rather than a following string. So we could use

```
" MYFILE.SEQ" $FLOAD
```

instead of the example above. In practice, this is more like to be used if the file name has been stored in a variable or inside a compiled word.

```
INCLUDE ( <name> --- )
```

This is equivalent to FLOAD and is normally used inside FLOADed files.

```
$INCLUDE ( addr --- )
```

This is a version of INCLUDE that takes an address rather than the input string.

## 3.10 Converting block files to/from ASCII files

There are a couple of words defined in BLK.SEQ that will let you convert block files (.BLK) files to ASCII (.SEQ) and vice versa. To load these words, use

```
FLOAD MYFILE.SEQ
```

Then, for example, to convert blocks 1 to 10 to of the file MYFILE.BLK to MYFILE.SEQ, you should first load the original file using

```
OPEN  MYFILE.BLK
```

and then convert it using

```
"  MYFILE.SEQ"  1  10  BLK>SEQ
```

This could then be compiled using

```
FLOAD  MYFILE.SEQ
```

To convert from an ASCII file to a block file use, for example,

```
"  MYFILE.SEQ"  SEQ>BLK
```

This will produce the file MYFILE.BLK with a two block header containing a title screen and a load screen. This could then be compiled using

```
OPEN  MYFILE.BLK   1  LOAD
```

`SEQ>BLK` uses the RAM disk when converting the file.

# 4 HiSoft FORTH Reference

## 4.1 Common FORTH Language Words

### 4.1.1 Glossary Notation

The glossary definitions are listed in ASCII alphabetical order. There is an entry for each of the **HiSoft FORTH** ST words. All cells in this implementation are 32 bit unless otherwise stated. The FORTH-83 standard specifies 16 bit, but this is really not very useful on a 32 bit processor.

The glossary is in the following format: on the first line is the name of the FORTH word, followed by a stack picture and then a pronunciation if necessary. The stack picture gives a before and after picture of what happens to the FORTH Data Stack. It reads from left to right, with the parameter next to the separator line on the left being the top of the stack before the word is executed and the right most parameter being the top of the stack afterwards. The next line gives a description of what the word does and what the parameters are used for. There is also a stack diagram to show the stack picture in more graphic detail where we think it might be useful.

The different abbreviations used in the stack pictures are as follows:

| | |
|---|---|
| 32b | 32 bit value |
| 16b | 16 bit value |
| 8b | 8 bit value |
| addr | 32 bit address |
| n | number |
| +n | positive number |
| d | double number (32 bit) |
| +d | positive double number |
| u | unsigned number |
| ud | unsigned double number |
| c | character |
| f | Boolean flag true = +n (-1); false = 0 |

# 4.2 HiSoft FORTH specific features

## 4.2.1 FORTH-83 standard and HiSoft FORTH

As we have said before **HiSoft FORTH** is largely compatible with other FORTHs but it has some features that are different to most other implementation because it returns on a 68000-based computer under GEM.

### 32 bit cell size

The Standard was written for 8/16 bit computers and assumes a 16 bit stack. This is wasteful on a processor like the 68000 so **HiSoft FORTH** uses a 32 bit stack. This affects common words like ! 'store', @ 'fetch' and 'comma'. Of course for higher level programming using 32 bits means that you can often ignore the possibility of arithmetic overflow.

Some words that required a double number but now use a single stack cell are as follows:

```
CONVERT ( +d1 addr1 --- +d2 addr2 )
<# ( --- )
#       ( +d1 --- +d2 )
#S ( +d --- 0 )
#> ( 32b --- addr +n )
```

### Strings

Strings are stored with a length byte and a trailing null character. So after

```
" HELLO"
```

is executed this will be stored as

| | 5 |
|---|---|
| "H" | 72 |
| "E" | 69 |
| "L" | 76 |
| "L" | 79 |
| "O" | 79 |
| | 0 |

This representation was chosen for ease of passing strings to the ST's operating system.

## Wildcard

When _ (underline) is used in a definition, then this is treated as a wild-card when the directory is searched. This is used to implement the `A:`, `B:`, `C:`, `D:` etc words without having a separate entry for each - the word `_:` is used for them all.

## Numbers

Hexadecimal numbers may be entered by preceding them with dollar (`$`). e.g.

```
$FF
```

places 255 on the stack.

To enter binary numbers use the prefix per-cent (`%`)

```
%101
```

places 5 on the stack.

## Glossary

`!` `( 32b addr --- )` 'store'

This word stores a 32 bit value at address in memory. In this implementation of FORTH variables are 4 bytes long or 32 bits.

`W!` `( 16b addr --- )` 'w-store'

This word stores a 16 bit value `16b` at address addr. This is the similar to `!` in a 16 bit FORTH.

`C!` `( 8b addr --- )` 'c-store'

This word stores a 8 bit value `8b` at address addr. This is the similar to Poke in the BASIC language.

`,` `( 32b --- )` 'comma'

Compile 32 bit number into the next available cell in the dictionary.

`W,` `( 16b --- )`

Compile 16 bit number into the next available cell in the dictionary.

`C,` `( 8b --- )`

Compile 8 bit number into the next available cell in the dictionary.

`( +n1 --- +n2 )` 'sharp'

Convert next digit unsigned number and add it to the beginning of the output string.

`#>` `( 32b --- addr +n )` 'sharp-greater'

End formatting of Formatted output string. Drops number remaining on the stack (usually 0) and leaves appropriate arguments for `TYPE`. The output string is generally held in memory just below `PAD`.

As the stack is 32 bit the 32b takes only one stack cell.

#S ( +n --- 0 ) 'sharp-s'

Convert all significant digits of unsigned number into the output.

As the stack is 32 bit the number +n takes only one stack cell. #S is typically used between <# and #>. Note that this is different from 16 bit FORTHs where this word requires a double.

' ( --- addr ) 'tick'

Use in the form

' <name>

Leave compilation address addr of <name>, which must be found within the the current search order.

In **HiSoft FORTH** the compilation address is the start of code.

* ( n1 n2 --- n3 ) 'times'

Take two numbers from the FORTH Data Stack and multiply them together leave the answer on the stack.

*/ ( n1 n2 n3 --- n4 ) 'times-divided'

Leave the ratio n4 = n1 * n2 / n3 where all are signed numbers.

*/MOD ( n1 n2 n3 --- quot mod ) 'times-divide-mod'

Leave the remainder n5 and quotient n4 of n1 * n2 / n3

+ ( n1 n2 --- n3 ) 'plus'

Leave the sum n3 of n1 + n2

+! ( 32b addr --- ) 'plus-store'

Add 32b to value at addr.

+W! ( 16b addr --- ) 'plus-w-store'

Add 16b to value at addr.

+C! ( 8b addr --- ) 'plus-c-store'

Add 8b to value at addr.

+LOOP ( n --- ) 'plus-loop'

Similar to LOOP but allows the increment to be changed. If you make the increment a negative value then the loop will count backwards.

+LOAD ( n --- )

Load relative to current block, (BLK + n).

+THRU ( n1 n2 --- )

Load a range of blocks n1 to n2 relative to current block.

e.g. 1 5 +THRU if this was loaded from block 10 would load from 11 to 15.

```
-       ( n1 n2 --- n3 ) 'minus'
```
Leave the difference of `n1 - n2`

```
-TRAILING ( addr +n1 --- addr +n2 )      'dash-trailing'
```
Reduce character count of a string at `addr` to omit trailing blanks.

```
.       ( n --- )       'dot'
```
Print signed number with one trailing blank.

```
."      ( --- )  'dot-quote'
```
Print all the following text until a " delimiter is reached.

Example:

```
HELLO ( --- )  ." Hello World!!!" ;
HELLO <ret> Hello World!!!
```

```
.(      ( --- )  'dot-paren'
```
Print the following string not including the delimiting `).`

Example:

```
.( Hello World!!!) <ret> Hello World!!!
```

```
.R      ( n1 n2 --- )
```
Print number `n1` right aligned in a field of `n2` characters wide.

```
.S      ( --- )
```
Displays the entire stack without changing it.

e.g.

```
        1 2 3 .S
```

would produce the following output:

```
TOS                         top of stack
[       3 ]
[       2 ]
[       1 ]
EMPTY                       bottom of stack
```

numbers in brackets are stack cells.

```
/       ( n1 n2 --- n3 ) 'divide'
```
Divide `n1` by `n2` leave quotient `n3` on stack.

```
/MOD ( n1 n2 --- quot mod )
```
Leave the remainder and signed quotient of `n1 / n2`, with the same sign as `n1`.

`: ( --- )`

Use in the form

```
: <name> .... ;
```

Begin compiling colon definition with name `<name>`.

`; ( --- )`

End colon definition.

`;S ( --- )`

Stop interpretation of a screen.

`< ( 32b --- 32b )`

Begin formatting a number on the stack into a string.

`<BUILDS ( --- )`

Make a header for `DOES>`.

`<MARK ( --- addr )`

Used at the destination of a backward branch. `addr` is typically only used by `<RESOLVE` to compile a branch address.

`<RESOLVE ( addr --- )`

Used at the source of a backward branch after either `BRANCH` or `?BRANCH` . Compiles a branch address using `addr` as the destination address.

`>MARK ( --- addr )`

Used at the source of a forward branch. Typically used after either `BRANCH` or `?BRANCH`. Compiles space in the dictionary for a branch address which will later be resolved by `>RESOLVE`.

`<< ( 32b n --- )`

Shift left `32b` value on stack by `n` bits.

`>> ( 32b n --- )`

Shift right 32b value on stack by `n` bits.

`>< ( 16b1 --- 16b2)`

Swap the high and low bytes within `16b1`.

`>RESOLVE ( addr --- )`

Used at the destination of a forward branch. Calculates the branch address (to the current location in the dictionary) using `addr` and places this branch address into the space left by `>MARK`.

`>BODY ( addr1 --- addr2 )` 'to-body'

`addr2` is the parameter field address corresponding to the compilation address `addr1`.

`>NAME ( addr1 --- addr2 )` 'to-name'

`addr2` is the name field address corresponding to the compilation address `addr1`.

`>LINK ( addr1 --- addr2 )` 'to-link'

`addr2` is the link field address corresponding to the compilation address `addr1`.

`>R ( 32b --- )`

Push a number from the Data Stack to the Return Stack. This should be balanced with a pull `R>` within the same definition.

`>WR ( 16b --- )`

Push a 16 bit number from the Data Stack to the Return Stack. This should be balanced with a pull `WR>` within the same definition. This is not a standard FORTH word - it is a HiSoft FORTH extension to provide a facility that would otherwise not be available because of the 32 bit word size.

`@ ( addr --- 32b )` 'fetch'

This word fetches a 32 bit value from the address on the stack.

`W@ ( addr --- 16b )` 'word-fetch'

This word fetches a 16 bit value `16b` from the address on the stack. This is similar to `@` in a 16 bit FORTH.

`C@ ( addr --- 8b )` 'c-fetch'

This word fetches a 8 bit value `8b` from the address on the stack. This is similar to Peek in BASIC language.

`= ( n1 n2 --- f )`

Leave a true flag if `n1` is equal to `n2`.

`> ( n1 n2 --- f )`

Leave a true flag if `n2` is greater than `n1`.

`< ( n1 n2 --- f )`

Leave a true flag if `n2` is less than `n1`.

`<> ( n1 n2 --- f )`

Leave a true flag if `n1` is not equal to `n2`.

`0= ( n --- f )`

Leave a true flag if `n` is equal to zero.

`0> ( n --- f )`

Leave a true flag if n is greater than zero.

`0< ( n --- f )`

Leave a true flag if n is less than zero.

`1+` `( d1 --- d2 )`

Add one to the top stack value.

`1-` `( d1 --- d2 )`

Subtract one to the top stack value.

`2+` `( d1 --- d2 )`

Add two to the top stack value.

`2-` `( d1 --- d2 )`

Subtract two to the top stack value.

`2*` `( d1 --- d2 )`

`d2` is the result of arithmetically shifting `d1` left one bit.

`2/` `( d1 --- d2 )`

`d2` is the result of arithmetically shifting `d1` right one bit.

`ABORT ( --- )`

Clear data and return stacks and return control to the keyboard without issuing an `OK`.

`ABORT"` `( flag --- )` 'abort-quote'

Used in the form

```
flag ABORT" ccc"
```

When later executed, if the `flag` is true the characters `ccc`, delimitated by " (close-quote), are displayed and then `ABORT` is executed. If the `flag` is false then the `flag` is dropped and execution continues.

`ABS` `( d --- ud )`

Remove the sign from the top of stack value and leave the absolute value.

`AGAIN ( --- )`

End of loop structure, always loop back to `BEGIN`.

`ALLOT ( d --- )`

Set aside `d` bytes in the dictionary starting at `HERE`. The address of next available dictionary pointer (`DP`) is updated.

`AND` `( n1 n2 --- n3 )`

Leave the bitwise logical AND of `n1` and `n2` as `n3`.

`ALSO ( --- )`

The transient vocabulary becomes the first vocabulary in the resident portion of the search order. Up to the last six resident vocabularies will also be reserved, in order, forming the resident search order.

PREVIOUS ( --- )

The transient vocabulary is replaced by the first vocabulary in the resident portion of the search order. The last six resident vocabularies are moved up, in order, forming the resident search order.

ARRAY ( n --- addr )

Create an array of n long words.The first element of an array starts at 0.

For example:

```
10 ARRAY ELEMENT          creates an array of 10 elements
123 2 ELEMENT !           stores 123 in element 2 of array
2 ELEMENT @               fetches element 2 to stack
2 ELEMENT ? 123           prints element 2, which is 123
```

WARRAY ( n --- addr )

Create an array of n words. See ARRAY.

CARRAY ( n --- addr )

Create an array of n bytes. See ARRAY.

ASCII ( --- )

Get following ASCII character on stack.

ASK ( --- )

System input routine, gets a string to input buffer includes history buffer.

ASSEMBLER ( --- )

This vocabulary contains a FORTH 68000 assembler.

AT ( x y --- )

Move cursor to column x and row y.

AUX: ( --- )

Vector output to the modem port.

BASE ( -- addr )

User variable containing current I/O radix, in the range 2-72.

BEGIN ( --- )

Marks the beginning of a loop structure.

e.g.

```
BEGIN ... AGAIN                    (loop always)
BEGIN .. f WHILE ... REPEAT        (loop while f is true)
BEGIN ... f UNTIL                  (loop while f is false)
```

BLANKS ( addr n --- )

Similar to fill but preset with a blank space character. We could use it to wipe our buffer with

```
buffer.addr 1024 BLANKS
```

BLK ( --- addr )

User variable containing the number of the mass storage block being interpreted as the input stream. If the value of BLK is zero the input stream is taken from the text input buffer.

BLOCK ( u --- addr )

addr is the address of the assigned buffer of the first byte of block u. If the block occupying the buffer is not block u and as been updated it is transferred to mass storage before assigning the buffer. If the block u is not already in memory, it is transferred from mass storage into an assigned block buffer. A block may not be assigned to more then one buffer. If u is not an available block number, an error condition exists. Only data within the last buffer referenced by BLOCK or BUFFER is valid. The contents of a block buffer mast not be changed unless the change may be transferred to mass storage.

BODY> ( addr1 --- addr2 ) 'from-body'

addr2 is the compilation field address corresponding to the parameter field address addr1.

BUFFER ( u --- addr )

Assigns a block buffer to block u. addr is the address of the first byte of the block within its buffer. This function is the same as BLOCK except that if it is not in memory it may not be transferred from mass storage. The contents of the buffer assigned to block u by BUFFER are unspecified.

CASE ( flag --- )

Use in the form

```
flag  CASE
        n1  OF  .....  ENDOF
        n2  OF  .....  ENDOF
        n3  OF  .....  ENDOF
        n4  OF  .....  ENDOF
        ........ (default  code)          ENDCASE
```

Start of a case block select structure. This will do multiple tests to see if any of the OF tests match flag. If there is a match then it will execute the FORTH words between OF and ENDOF of the true test. If there are no matches it will execute the code between the last ENDOF and ENDCASE.

CFA ( PFA --- CFA)

Convert the parameter field address of a definition to its code field address.

CLREOL ( --- )

Clear from cursor to the end of line.

CLREOP ( --- )

Clear from cursor to the end of the screen.

`CMOVE ( addr1 addr2 n --- )`

`CMOVE` moves a block of memory from one place to another. Where `addr1` is the source block, `addr2` is the destination block and `n` is the number of bytes to move.

`COMPILE ( --- )`

Used in the form

```
COMPILE <name>
```

Compile the code field address (`CFA`) of the non-immediate word `<name>` which follows into the dictionary upon execution of the current definition.

`CON: ( --- )`

Vector output to the console.

`CONSTANT ( n -- )`

Make a FORTH word that leaves the number `n` when it executes.

`CONVERT ( +d1 addr+1 --- +d2 addr2 )`

Convert string at `addr+1` to double number and add value into `+d1` leaving result +d2, `addr2` is address of first non-convertible character.

`COUNT ( addr -- addr+1 count )`

If addr contains a count byte followed by some text, then leave the addr of the text (`addr+1`) and the count on the stack. This can be used by `TYPE` to print a variable to the screen.

`CR ( --- )`

Do a carriage return and line feed.

`CREATE ( --- )`

`CREATE` makes a header for new FORTH words, it is used by `:` and other FORTH defining words. The FORTH-83 version is different, see **Section 4.4**.

`CSP ( --- addr )`

A System variable temporarily storing the stack pointer position, for compilation error checking.

`!CSP ( --- )`

Save the stack position in `CSP`. Used as part of compiler security.

`?CSP ( --- )`

Issue an error message if stack position differs from value saved in CSP.

`DECIMAL ( --- )`

Set decimal number base.

```
DEFER ( --- )
```

Creates a deferred execution word.

```
e.g.    DEFER CLS
        ' PAGE IS CLS
```

would firstly create a deferred execution word `CLS` the code would then be patched into `CLS` by `IS`.

```
DEFINITIONS ( --- )
```

Select the transient vocabulary as the current vocabulary into which subsequent definitions will be added.

```
DEPTH ( --- +n )+n
```

is the number of 32 bit values contained in the Data Stack before `+n` was placed on the stack.

```
DO      ( n1 n2 --- )
```

Start of `DO-LOOP` structure, with a loop limit of `n1` and an initial index of `n2`. The index is incremented by 1 until it equals or exceeds the limit. Executes words between `DO` and `LOOP` on each time through the loop.

e.g.

```
        4 0 DO ... I ...        LOOP

(loop 4 times, I equals 0 1 2 3)

        4 0 DO ... I ... 2 +LOOP

(loop 2 times, I equals 0 2)

        0 4 DO ... I ... -1 +LOOP

 (loop 4 times, I equals 4 3 2 1)
```

```
DOES> ( --- addr ) (compiling)
```

Defines the execution-time action of a word created by a high level defining word. Used in the form

```
        : <namex> ... <BUILDS .... DOES> ... ;
```

and then

```
        <namex> <name>
```

Marks the termination of the defining part of the defining word `<namex>` and then begins the definition of the execution-time action for words that will later be defined by `<namex>`. When `<name>` is later executed, the address of `<name>`'s parameter field is placed on the stack and then the sequence of words between `DOES>` and `;` are executed.

```
DP      ( --- addr )
```

User variable containing the address of the dictionary pointer.

```
DROP    ( 32b --- )
```

Drop top stack value.

```
DUP     ( 32b --- 32b 32b )
```

Duplicate top stack value.

```
?DUP    ( 32b --- 32b 32b ) F83std
-DUP    ( n --- n n )       fig
```

Duplicate the top stack value only if non-zero.

```
EDITOR  ( --- )
```

This vocabulary contains the FORTH editor, the editor is loaded from disk.

```
ELSE    ( --- )
```

Optional word used between IF and THEN, if the flag tested by IF is false then execute the words between ELSE and THEN.

```
EMIT    ( c --- )
```

Transmit ASCII character c to the selected output device. The user variable OUT is incremented for each character output.

```
EMPTY-BUFFERS    ( --- )
```

Erase and un-assign all block buffers.

```
ENDCASE ( --- )
```

Marks the end of a CASE select block structure.

```
ENDOF ( --- )
```

Marks the end of an OF select block.

```
ERASE ( addr n --- )
```

Similar to BLANKS but this time preset with the value zero or ASCII NULL. We could erase our buffer to contain all zeros with

```
buffer.addr  1024  ERASE
```

```
EXECUTE ( CFA --- )
```

This word executes a FORTH CFA (code field address) on the stack.

```
EXPECT  ( addr count --- )
```

Transfer characters from the terminal to address, until a Return or the count of characters have been received. An ASCII NULL is added to the end of the text.

`FILL ( addr n char --- )`

You can use `FILL` to fill an area of memory with a character value. If you had a buffer of 1024 bytes length and you wanted to fill it with the character `A` (ASCII 65) then you could use

```
buffer.addr 1024 65 FILL
```

to do the job.

`FIND ( addr1 --- addr2 n )`

For a string with a count byte at `addr1` search for a matching word name using the transient and resident search orders. If the word is not found then `addr2` is the string address `addr1` and `n` is zero. If found `addr2` is the matching word's compilation address and if the word is immediate `n` is set to 1. If the word is not immediate then the word is set to -1 (true).

`FLUSH ( --- )`

The same as `SAVE-BUFFERS` but also un-assigns all block buffers.

`FORGET ( --- )`

Used in the form

```
FORGET <name>
```

Delete from the dictionary `<name>` and all words added to the dictionary after `<name>` regardless of the vocabulary. Failure to find `<name>` is an error condition. An error condition also exists upon implicitly forgetting a vocabulary (due to its definition after `<name>`).

`FORTH ( --- )`

The name of the primary vocabulary. Execution makes FORTH the transient vocabulary, the first in the search order, and thus replaces the previous transient vocabulary.

`FORTH-83 ( --- )`

The same as `FORTH` but chains FORTH-83 standard words into the top of the FORTH search order. This makes available a FORTH-83 standard system.

`FNAME ( --- addr )`

Pathname used by system to find `FORTH.BLK`.

`GEM ( --- )`

This vocabulary contains ATARI ST GEM calls for use from high level FORTH.

`H. ( n --- )`

Print top of stack number in the hexadecimal base.

`HERE ( --- addr )`

Leave the address of the next available dictionary location.

HEX ( --- )

Set hexadecimal (base 16) number base. You can also prefix hexademical numbers by using $ in ordinary decimal mode. e.g.

```
$FF     is 255
```

HOLD ( c --- )

Insert ASCII character into formatted output string. e.g.

```
46 HOLD
```

inserts a decimal point.

HOME ( --- )

Move the cursor to the top left hand side of screen.

I ( --- n )

DO-LOOP counter.

J ( --- n )

Nested DO-LOOP counter.

K ( --- n )

Double nested DO-LOOP counter.

IF ( f --- )

Test the flag f on the stack, if it is true execute the words between IF and THEN.

IS ( addr --- )

Used to patch a code addr into a deferred execution word created by DEFER. See DEFER for example of usage.

KEY ( --- c )

Leave the ASCII value of the next terminal key struck.

L>NAME ( addr1 --- addr2 ) 'link-to-name'

addr2 is the name field address corresponding to the link field address addr1.

LEAVE ( --- )

Force termination of a DO-LOOP.

LFA ( PFA --- LFA)

Convert the parameter field address of a definition to its link field address.

LINK> ( addr1 --- addr2 ) 'from-link'

addr2 is the compilation field address corresponding to the link field address addr1.

LOAD ( u --- )

Interpret screen u as if it were keyboard input. When finished return control to the keyboard.

LOGOUT ( --- )

Flush open files to disk and clear disk map.

LOOP ( --- )

End of DO-LOOP loop structure. Loops back to DO.

M* ( n1 n2 --- d)

A mixed magnitude math operation which leaves the double number signed product d of two singed numbers.

M/ ( d n --- remainder quotient)

A mixed magnitude math operator which leaves the signed remainder and signed quotient from a double number dividend d and divisor n. The remainder takes its sign from the dividend.

MACRO ( --- )

Makes previously defined word into a MACRO definition. The constraints on macro words are that must be less than 32 bytes long and must not contain any relative code.

e.g. To define a MACRO for 2*

```
: 2* DUP + ; MACRO
```

MAKEFILE ( addr --- )

Make new file with filename at addr on stack.

e.g.

```
" MYFILE.BLK" MAKEFILE
```

would make a new file called MYFILE.BLK.

MAX ( n1 n2 --- n3 )

Leave the greater of the two top stack items and discard the other.

MIDI: ( --- )

Vector output to the midi port.

MIN ( n1 n2 --- n3 )

Leave the smaller of the two top stack items and discard the other.

MINUS ( n1 --- n2 ) fig

Leave the two's complement of a number.

MOD ( n1 n2 --- mod )

Leave the remainder of n1 / n2, with the same sign as n1.

MORE ( n --- )

Add n blocks to current disk stream.

N>LINK ( addr1 --- addr2 ) 'name-to-link'

addr2 is the link field address corresponding to the name field address addr1.

NAND ( 32b1 32b2 --- 32b3)

32b3 is the one's complement of the logical AND of 32b1 and 32b2.

NAME> ( addr1 --- addr2 ) 'from-name'

addr2 is the compilation field address corresponding to the name field address addr1.

NEGATE ( n1 --- n2 ) F83std

Leave the two's complement of a number.

NFA ( PFA --- nfa)

Convert the parameter field address of a definition to its name field address.

NIP ( n1 n2 --- n2)

Drop second stack cell value.

NOT ( n1 --- n2 )

Leave the one's complement of n1 as n2.

NOR ( 32b1 32b2 --- 32b3)

32b3 is the one's complement of the logical OR of 32b1 and 32b2.

NXOR ( 32b1 32b2 --- 32b3)

32b3 is the one's complement of the logical XOR of 32b1 and 32b2.

O. ( n --- ) 'O- dot'

Print top of stack number in base 8.

OF ( n --- )

Used with ENDOF, between CASE and ENDCASE. OF will test for a match between n and the flag before CASE. If there is a match it will execute the FORTH code up to the following ENDOF. If not then execution will continue at the next OF or ENDCASE if there are no more OFs.

```
OK      ( --- )
```
System prompt. This word is deferred and can be changed.

e.g.

```
: PROMPT
        .S 'OK ;

' PROMPT ' OK 2+ !
```

would change prompt to user defined word `PROMPT`.

```
' 'OK ' OK 2+ !
```

would change back to system prompt.

```
ONLY    ( --- )
```
Select just the `ONLY` vocabulary as both the transient vocabulary and the resident vocabulary in the search order.

```
OR      ( n1 n2 --- n3 )
```
Leave the bitwise logical OR of `n1` and `n2` as `n3`.

```
ORDER   ( --- )
```
Display the vocabulary names forming the search order in their present search order sequence. Then show the vocabulary \ into which new vocabularies will be placed.

```
OVER    ( 32b1 32b2 --- 32b1 32b2 32b1 )
```
Duplicate the second form top value on the stack and place it on top of the stack.

```
PAD ( --- addr )
```
Leave a pointer to the first byte of a floating scratch pad area.

```
PAGE    ( --- )
```
Clear screen and home cursor.

```
PFA     ( nfa --- pfa)
```
Convert the name field address of a definition to its parameter field address.

```
PICK    ( +n --- 32b )
```
Leave a copy of the +nth stack location, not counting `+n` itself.

```
PNAME ( --- addr )
```
Pathname used by system to find `FORTH.BLK`.

```
PRT:    ( --- )
```
Vector output to the printer.

```
QUIT    ( --- )
```
Clear Return Stack and return control to the keyboard. No message is displayed.

`REPEAT ( --- )`

End of loop structure used with `WHILE`, loop back to `BEGIN` if `WHILE` test is true.

`ROT ( 32b1 32b2 32b3 ---- 32b2 32b3 32b1 )`

Rotate the third value on the stack to the top of the stack.

`ROLL ( +n -- 32b )`

The +nth stack value, not counting `+n` itself is first removed and then transferred to the top of the stack, moving the remaining values into the vacated position.

`RSP ( --- addr )`

A System variable temporarily storing the Return Stack pointer position.

`!RSP ( --- )`

Save the Return Stack position in `RSP`.

`@RSP ( --- )`

Restore Return Stack pointer from `RSP`.

`R> ( --- 32b )`

Pull a number from the Return Stack to the Data Stack. See `>R` and `R`.

`R@ ( --- 32b ) F83std`

Copy a number from the Return Stack to the Data Stack. See also `>R` and `R`.

`R ( --- 32b )`

Copy a number from the Return Stack to the Data Stack. See also `>R` and `R`.

`WR> ( --- 16b ) Non-standard`

Pull a 16 bit number from the Return Stack to the Data Stack. See also `>WR`.

`RP@ ( --- addr )`

Return the address of the Data Stack.

`RP! ( --- )`

Initialise the Return Stack from user variable `R0`.

`R/W ( buff_addr blk_number R/Wflag --- )`

Reads (if `R/Wflag` is 1) or writes (if `R/Wflag` is 0) 1024 bytes from block blk_number to/from memory address `buff_addr`. The current stream is used.

`S->D ( 16b --- 32b )`

Convert a sign extended 16 bit number into a sign extended 32 bit number.

`SAVE-BUFFERS ( --- )`

The contents of all block buffers marked as updated are written to their corresponding mass storage blocks. All buffers are marked as no longer modified, but may remain assigned.

`SCAN ( --- c scan )`

Return scan code and key value `c`.

`SEAL ( --- )`

Delete all occurrences of `ONLY` from the search order. The effect is that only specified application vocabularies are searched.

`SIGN ( n --- )`

If signed number is less than zero insert minus sign at the beginning of a formatted output string.

`STRING$ ( n --- )`

Create a string variable of `n` bytes.

`SYSVAR ( n --- addr )`

Create a system variable. Similar to `USER` in fig, see also `USER` in `TASK` vocabulary.

`SP@ ( --- addr )`

Return the address of the Data Stack.

`SP! ( --- )`

Initialise the Data Stack from user variable `S0`.

`SPACE ( --- )`

Type one space.

`SPACES ( +n --- )`

Type `+n` spaces.

`SPAN ( --- addr )`

User variable containing the number of characters received and stored by the last execution of `EXPECT`.

`STATE ( --- addr )`

System variable whose value is non-zero when compilation is occurring and false (zero) when interpreting.

`STRLEN ( addr --- addr n)`

Find length `n` of zero terminated string addr.

`SWAP ( 32b1 32b2 --- 32b2 32b1 )`

Swap the top two values on the stack.

`THEN ( --- )`

used with `IF` marks then end of a conditional FORTH block.

`THRU ( n1 n2 --- )`

Load a range of blocks from `n1` to `n2`.

`TIB ( --- addr )`

Leave a pointer to the first byte of the terminal input buffer. The buffer length is 80 characters.

`TOGGLE ( addr --- b)`

Complement the byte value of addr by the bit pattern `b`.

`TOS ( --- )`

This vocabulary contains ATARI ST TOS calls for use from high level FORTH.

`TUCK ( n1 n2 --- n1 n1 n2)`

Duplicate second stack cell value.

`TYPE ( addr count --- )`

Transmit `count` characters from `addr` to the selected output device.

`U. ( u --- )`

Print unsigned number `u`.

`U.R ( u n --- )`

Print unsigned number `u` in a field of `n` characters.

`U* ( u1 u2 --- ud)`

`ud` is the unsigned product of `u1` times `u2`. All values and arithmetic are unsigned.

`U/ ( ud u1 --- remainder quotient)`

Leave the unsigned remainder and unsigned quotient from the unsigned double dividend `ud` and unsigned divisor `u1`.

`UNDER ( n1 n2 --- n2 n1 n2)`

Copy top of stack under second stack cell.

`UNTIL ( f --- )`

End of loop structure, test flag f on stack loop back to `BEGIN` if true.

`UPC ( --- )`

Use UPC ON to cause all user input to be converted to upper case.

`UPDATE ( --- )`

Mark the currently valid block as modified, so that if the buffer is needed the block will be written to mass storage.

`UPPER ( addr n --- )`

Convert string of `n` bytes at address `addr` to uppercase.

`VARIABLE ( n -- )`

Make a FORTH word that leaves an address on the stack that points to a space in memory that is initialised to the value `n`.

`VOC-LINK ( --- addr )`

A system variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields.

`W* ( 16b 16b --- 32b)`

16 bit multiply, faster than `*` . This word uses `MULS` to give a fast multiply.

`W/ ( 32b 16b --- remainder quotient)`

Divide `32b` by `16b` and leave quotient and remainder. This word uses the DIVS opcode to give a fast divide.

`WHILE ( f --- )`

Test flag f on stack, if it is true then execute the words between `WHILE` and `REPEAT`.

```
WORD    ( c --- addr )  F83std
        ( c --- )       fig
```

Parses the next word delimitated by `c` or the end of the input stream and stores it with its count byte at address. If the string is longer than 255 characters, the count is unspecified. If the input stream is already exhausted then the count equals zero. The character count in `>IN` (F83std) and `IN` (fig) is updated to indicate the character after the final delimiter. In FIG-FORTH the string is left at `HERE`.

`WORDS ( --- )`

Display the word names in the transient vocabulary, starting with the most recent definition.

`WTOGGLE ( addr --- w)`

Complement the word value at `addr` by the bit pattern `w`.

`XOR ( n1 n2 --- n3 )`

Leave the bitwise logical XOR of `n1` and `n2` as `n3`.

`[ ( --- )` (compiling)

Stop compiling input text and begin executing.

`] ( --- )` (compiling)

Stop executing input text and begin compiling.

`['] ( --- addr )` (compiling) F83std

Use in the form

```
['] <name>
```

Immediate word to compile the compilation address of `<name>` as a literal within a definition. The address is left on the stack upon execution of the definition.

`[COMPILE] ( --- )`

Used in the form

```
COMPILE] <name>
```

Causes the word `<name>` that follows to be compiled into the current definition even if it is immediate.

## 4.3 FORTH-79 Standard

The following words are changed from fig-FORTH. These words are brought to the top of the search order when you execute `79-STANDARD`.

The vocabulary 79-STANDARD is chained to the FORTH vocabulary. The word `FORTH` is redefined to select the 79-STANDARD vocabulary. To return to the default system FORTH vocabulary execute

```
ONLY FORTH DEFINITONS
```

`79-STANDARD` also changes the current vocabulary to 79-STANDARD.

`?DUP ( n --- n (n))`

Duplicate top stack value only if non-zero. This word is the same as `-DUP` in fig-FORTH.

`>IN ( --- addr)`

System variable containing character offset into input buffer. This word is the same as `IN` in fig-FORTH.

`BLANK ( addr n --- )`

Write `n` blank characters to memory starting at `addr`. This is the same as fig-FORTH word `BLANKS`.

`CONVERT ( n1 addr1 --- n2 addr2)`

Convert string at `addr1+1` to number and add number into `n1` leaving result `n2`. `addr2` is address of first non-convertible character. This is the same as `(NUMBER)` in fig-FORTH.

The FORTH-79 standard actually uses double numbers with this word, this was not done in this implementation as the stack is 32 bit, so a single stack cell was used instead.

`CREATE ( --- addr)`

Create a header that leaves an address when executed. The fig-FORTH system `CREATE` is different in 2 ways. Firstly it makes only a header and can not be executed without following code, secondly because of this it does not leave a address on the stack.

In FORTH-79 and FORTH-83, CREATE can be used to make defining words with DOES> e.g.

```
: VARIABLE CREATE 0 , DOES> ;
```

this would define variables like the FORTH-79 VARIABLE . In fig-FORTH CREATE would be replaced by <BUILDS. e.g.

```
: VARIABLE <BUILDS , DOES> ;
```

this would define fig-FORTH variables that would be initialised from the stack.

FIND ( --- addr)

Find the code field address of name in dictionary. If name can not be found leaves a zero on stack instead of addr. Used in the form:

```
FIND <name>
```

NEGATE ( n --- -n)

Reverse the sign of the top of stack value, (two's complement). This is the same as the FIG-FORTH word MINUS.

PICK ( n1 --- n2)

Copy n1th item on the stack to the top. e.g.

```
1 PICK  is equivalent to DUP
2 PICK  is equivalent to OVER
```

ROLL ( n1 --- )

Rotate n1th item to top of stack. e.g.

```
2 ROLL  is equivalent to SWAP
3 ROLL  is equivalent to ROT
```

R@ ( --- n)

Copy top of Return Stack to top of Data Stack. The same as FIG-FORTH word R .

U/MOD ( ud u --- rem quot)

Divide double number by single giving unsigned remainder and quotient. All values are unsigned. This word is the same as FIG-FORTH word U/ .

VARIABLE ( --- addr)

Create a 4 byte variable, which returns its address when executed. e.g.

```
VARIABLE NAME
```

This is different to the FIG-FORTH word VARIABLE as it does not require a number on the stack to initialise variable.

```
WORD   ( c --- addr)
```

Read the next word from the input buffer using `c` as delimiter, or until null. Leave address of length byte of word. This similar to `WORD` in FIG-FORTH, but the fig `WORD` puts the string at `HERE` and does not leave the address on the stack. The addr returned by `WORD` is the same as `HERE`.

# 4.4 FORTH-83 Standard

The purpose of the FORTH-83 Standard is to allow for portability of FORTH-83 Standard Programs in source form among FORTH-83 Standard Systems. To comply with the Standard a FORTH implementation must include the required word set in the vocabulary FORTH, after executing the word `FORTH-83`.

## 4.4.1 The Required word set

The words of the required word set are grouped to show like characteristics.

Nucleus layer

```
! * */ */MOD + +! - / /MOD 0< 0= 0> 0> 1+ 1-2+ 2- 2/ < = >
>R ?DUP @ ABS AND C! C@ CMOVE CMOVE>COUNT D+ D< DEPTH
DNEGATE DROP DROP DUP EXECUTE EXITFILL I J MAX MIN MOD
NEGATE NOT OR OVER PICK R> R@ROLL ROT SWAP U< UM* UM/MOD
XOR
```

Device layer

```
BLOCK BUFFER CR EMIT EXPECT FLUSH KEY SAVE-BUFFERS SPACES
PACES TYPE UPDATE
```

Interpreter layer

```
# #> #S #TIB ' ( -TRAILING . .( <# >BODY >IN ABORT BASE BLK
CONVERT DECIMAL DEFINITIONS FIND FORGET FORTH FORTH-83 HERE
HOLD LOAD PAD QUIT SIGN SPAN TIB U. WORD
```

Compiler layer

```
+LOOP , ." : ; ABORT" ALLOT BEGIN COMPILE CONSTANT CREATE
DO DOES> ELSE IF IMMEDIATE LEAVE LITERAL LOOP REPEAT STATE
THEN UNTIL VARIABLE VOCABULARY WHILE [ ['] [COMPILE] ]
```

## 4.4.2 FORTH-83 standard and HiSoft FORTH

The Standard was written for 8/16 bit computers and assumes a 16 bit stack. This is wasteful on a processor like the 68000 so **HiSoft FORTH** uses a 32 bit stack.

Some words that required a double number but now use a single stack cell are as follows:

```
CONVERT ( +d1 addr1 --- +d2 addr2 )
<# ( --- )
#       ( +d1 --- +d2 )
#S ( +d --- 0 )
#> ( 32b --- addr +n )
```

## 4.4.3 FORTH-83 Words

The following words are changed from FIG-FORTH and FORTH-79 standard. These words are brought to the top of the search order when you execute `FORTH-83` in compliance with the FORTH-83 standard.

The vocabulary FORTH-83 is chained to the FORTH vocabulary. The word `FORTH` is redefined to select the FORTH-83 vocabulary. To return to the default system FORTH vocabulary execute

```
ONLY FORTH DEFINITONS
```

FORTH-83 also changes the current vocabulary to FORTH-83.

Words unchanged from the FORTH-79 standard are :

```
->IN ?DUP BLANK CONVERT CREATE R@ WORD VARIABLE
```

`.(` `( --- )` 'dot-paren'

Used in the form:

```
.( ccc)
```

The characters `ccc` up to but not including the delimiting `)` are displayed. The blank following `.(` is not part of `ccc`. This may be used to include double quote characters in strings.

`."` `( --- )` 'dot-quote'

Used in the form:

```
." ccc"
```

Later execution will display the characters `ccc` up to but not including the delimiting `"` . The blank following `."` is not part of `ccc`.

`'` `( --- addr)` 'tick'

Used in the form:

```
' <name>
```

`addr` is the compilation address of `<name>`. An error condition exists if `<name>` is not found in the currently active search order.

`[']  ( --- addr)  'bracket-tick'`

Used in the form:

```
' <name>
```

Compiles the compilation address addr of `<name>` as a literal. When the colon definition is later executed the addr is left on the stack. An error condition exists if `<name>` is not found in the currently active search order.

`/  ( n1 n2 --- n3)  'divide'`

`n3` is the floor of the quotient of `n1` divided by the divisor `n2`. An error condition results if the divisor is zero.

`*/  ( n1 n2 n3 --- n4)  'times-divide'`

`n1` is first multiplied by `n2` producing an intermediate 64 bit result. `n4` is the remainder and `n5` the floor of the quotient of the intermediate 64 bit result divided by the divisor `n3`.

`/MOD  ( n1 n2 --- n3 n4)  'divide-mod'`

`n3` is the remainder and `n4` the floor of the quotient of `n1` divided by the divisor `n2`. `n3` has the same sign `n2` or is zero. An error condition results if the divisor is zero.

`MOD  ( n1 n2 --- n3)`

`n3` is the remainder after dividing `n1` by the divisor `n2`. `n3` has the same sign `n2` or is zero. An error condition results if the divisor is zero.

`ABORT"  ( flag --- )  'abort-quote'`
`( --- ) (compiling)`

Used in the form:

```
flag ABORT" ccc"
```

When later executed, if `flag` is true the characters `ccc`, delimited by `"` , are displayed and then a system dependent error abort sequence, including the function of `ABORT` , is performed. If `flag` is false the `flag` is dropped and execution continues.

`CMOVE>  ( addr1 addr2 u --- )  'c-move-up'`

Move `u` bytes beginning at address `addr1` to `addr2`. The move begins by moving the byte at `addr1+u-1` to `addr2+u-1` and proceeds to successively lower bytes for `u` bytes. If `u` is zero nothing is moved.

```
DO      ( n1 n2 --- )
        ( --- sys)      (compiling)
```

Used in the form:

```
        DO ... LOOP
    or
        DO ... n +LOOP
```

Begins a loop which terminates based on control parameters. The loop index begins at `n2` and terminates based on the limit `n1`. The loop is always executed at least once.

```
FIND    ( addr1 --- addr2 n)
```

`addr1` is a counted string. The string contains a word name to be located in the currently active search order. If the word is not found, `addr2` is the string address `addr1` and `n` is zero. If the word is found, `addr2` is the compilation address and `n` is set to one for immediate words and minus one for non-immediate words.

```
LAST    ( --- addr)
```

Leaves the compilation addr of the latest definition. See FIG-FORTH word `LATEST`.

```
LEAVE   ( --- )
        ( --- )  (compiling)
```

Transfer execution to just beyond the next `LOOP` or `+LOOP` . The loop is terminated and loop control parameters are discarded. May only be used in the form:

```
        DO ... LEAVE ... LOOP
```

or

```
        DO ... LEAVE ... +LOOP
```

`LEAVE` may appear in other control structures which are nested within the `DO-LOOP` structure. More than one `LEAVE` may appear within a `DO-LOOP`.

```
LOOP    ( --- )
        ( sys --- )     (compiling)
```

Increments the `do-loop` index by one. If the new index was incremented across the boundary between `limit-1` and the `limit` the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding `DO` . `sys` is balanced with its corresponding `DO` .

`+LOOP ( n --- )`
`( sys --- )` (compiling)

`n` is added to the loop index. If the new index was incremented across the boundary between `limit-1` and the `limit` the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding `DO` . `sys` is balanced with its corresponding `DO` .

`.NAME ( addr --- )`

Print the name of word with compilation address addr. See `ID.` .

`PICK ( +n --- 32b)`

`32b` is a copy of the `+n`th stack value, not counting `+n` itself. e.g.

`0 PICK` is equivalent to `DUP`

`1 PICK` is equivalent to `OVER`

`ROLL ( +n --- )`

The `+n`th stack value, not counting `+n` itself is first is removed and then transferred to the top of the stack, moving the remaining values into the vacated position. e.g.

`2 ROLL` is equivalent to `ROT`

`1 ROLL` is equivalent to `SWAP`

`0 ROLL` is a null operation.

`RP! ( addr --- )`

Initialise Return Stack with `addr`.

`SP! ( addr --- )`

Initialise Data Stack with `addr`.

`S>D ( 16b --- 32b)`

Convert 16 bit number to 32 bit number.

`TIB ( --- addr)`

The address of the input buffer. This buffer is used to hold characters when the input stream is coming from the current input device.

`#TIB ( --- addr)`

The address of a variable containing the number of bytes in the text input buffer.

`UM* ( u1 u2 --- ud)`

`ud` is the unsigned product of `u1` times `u2`. All values are unsigned. This is the same as `U*` in FIG-FORTH.

`UM/MOD ( ud u1 --- u2 u3)`

Leave the remainder u2 and the floored quotient u3 of ud divided by u1.

# 4.5 ONLY Words

These words are in the root vocabulary. They are a minimal word set that handles vocabulary switching the editor and some utilities.

Vocabularies are normally closed, that is they do not chain to other vocabularies. To chain a vocabulary use `CHAIN`. The search order of the system can be seen using `ORDER` .

There are 8 `CONTEXT` vocabulary slots, which are search when compiling or interpreting source code. The `CURRENT` vocabulary is used to compile new code into dictionary. The `ONLY` vocabulary is special and has its own slot.

All vocabularies are defined in the ONLY vocabulary. To FORGET a vocabulary you need to be in the ONLY vocabulary.

`QX ( n --- )`

Quick index, will index 64 screens starting from `n` . To get the most benefit from the quick index words it is best to create `.BLK` files in multiples of 64 screens.

`NX ( --- )`

The same as `QX` , but uses value in `SCR` system variable for `n` .

`BX ( --- )`

The same as `NX` , but uses value in `SCR - 64` to list previous 64 screens.

`SEAL ( --- )`

Removes `ONLY` vocabulary from search order, effectively sealing the system search order.

`CHAIN ( --- )`

Chains vocabulary to current vocabulary. This is similar to how a vocabulary would be defined in FIG. FORTH-83 and 79-STANDARD are both examples of chained vocabularies.

`SYSVEC ( --- )`

This word patches system vectors to trap errors back into FORTH. If you want to use a debugger such as **HiSoft** MONST or AMONST from **DevpacST** then you should not execute this word in your startup screen.

`SHRINK ( n --- )`

On startup **HiSoft FORTH** takes all available memory. `SHRINK` will return memory to TOS by only allocating `n` bytes of user dictionary space.

`DESKTOP ( --- )`

System word used by desk accessories.

`SLOT ( --- addr )`

Menu slot address used by desk accessories.

`BOOT ( --- )`

Used by the system at startup to boot from `FORTH.BLK` screen 1 .

`WORDS ( --- )`

List the word names in the first vocabulary of the currently active search order.

Keys that control `WORDS` are as follows:

`Ctrl-S` will pause listing

`Ctrl-Q` will restart listing

any other key will break listing.

`VLIST ( --- )`

List the names of definitions in the `CONTEXT` vocabulary.

`ORDER ( --- )`

Show current vocabulary search order. e.g.

```
        ORDER
Context: FORTH ONLY
Current: FORTH
```

The compiler would search `FORTH` and then `ONLY` and compile new code definitions into `FORTH`.

`FORGET ( --- )`

`VOCS ( --- )`

Prints list of vocabularies to screen. The System vocabularies are as follows:

| | |
|---|---|
| `ONLY` | Root vocabulary |
| `FORTH` | Main FORTH words |
| `FORTH-83` | FORTH-83 standard |
| `79-STANDARD` | FORTH-79 standard |
| `EDITOR` | Words used by the editor |
| `TOS` | Atari TOS words inc GEMDOS, BIOS and XBIOS. |
| `GEM` | Atari GEM words (the AES and VDI) |
| `TASK` | Multi-tasking words |
| `GRAPHIC` | Turtle graphics |
| `SHELL` | The Shell. See `SHELL.SEQ` |

# 4.6 Double Number Extension Set

You may enter double numbers by ending a number with a dot (.) e.g.

```
1234567890.
```

places this double number on the stack.

These words are provided to 64 bit double numbers.

`2!` `( 64b addr --- )` 'two-store'

`64b` is stored at addr.

`2@` `( addr --- 64b)` 'two-fetch'

`64b` is the value at addr.

`2CONSTANT` `( 64b --- )` `'two-constant'`

A defining word executed in the form:

```
64b 2CONSTANT <name>
```

Creates a dictionary entry for `<name>` so that when `<name>` is later executed, `64b` will be left on the stack.

`2DROP` `( 64b --- )` `'two-drop'`

`64b` is removed from the stack.

`2DUP` `( 64b --- 64b 64b)` `'two-dupe'`

Duplicate `64b`.

`2OVER` `( 64b1 64b2 --- 64b1 64b2 64b3)` 'two-over'

`32b3` is a copy of `32b1`.

`2ROT` `( 64b1 64b2 64b3 --- 64b1 64b2 64b3)` 'two-rote'

The top three double numbers on the stack are rotated, bringing the third double number to he top of the stack.

`2SWAP` `( 64b1 64b2 --- 64b2 64b1)` 'two-swap'

The top two double numbers are exchanged.

`2VARIABLE` `( 64b --- )` 'two-variable'

`( --- )` F79,F83

A defining word executed in the form:

```
64b 2VARIABLE <name>
```

A dictionary entry for `<name>` is created and 8 bytes are allotted in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initialising the contents of the variable which it creates. When `<name>` is later executed, the address of this parameter field is placed on the stack.

`D+` `( d1 d2 --- d3)`

`d3` is the sum of `d1` and `d2`.

`D+-` `( d1 n --- d2)`

Apply the sign of `n` to the double number `d1`, leaving it as `d2`.

`D-` `( d1 d2 --- d3)` 'd-minus'

`d3` is the result of subtracting `d2` from `d1`.

`D.` `( d --- )` 'd-dot'

The absolute value of `d` is displayed in a free field format. A leading negative sign is displayed if `d` is negative.

`D.R` `( d +n --- )` 'd-dot'

`d` is converted using the value of `BASE` and then displayed right aligned in a field `+n` characters wide. A leading negative sign is displayed if d is negative. If the number of characters required to display d is greater than `+n`, an error condition exists.

`D0=` `( d --- flag)` 'd-zero-equal'

`flag` is true if `d` is zero.

`D2/` `( d1 --- d2)` 'd-two-divide'

`d2` is the result of `d1` arithmetically shifted right one bit. The sign is included in the shift and remains unchanged.

`D<` `( d1 d2 --- flag)` 'd-less'

flag is true if `d1` is less than `d2`.

`D=` `( d1 d2 --- flag)` 'd-equal'

flag is true if `d1` equals `d2`.

`DABS` `( d --- ud)` 'd-absolute'

`ud` is the absolute value of `d`.

`DMAX` `( d1 d2 --- d3)` 'd-max'

`d3` is the greater of `d1` and `d2`.

`DMIN` `( d1 d2 --- d3)` 'd-min'

`d3` is the lesser of `d1` and `d2`.

`DNEGATE` `( d1 --- d2)` 'd-negate'

`d2` is the two's complement of `d1`.

`DU<` `( ud1 ud2 --- flag)` 'd-u-less'

flag is true if `ud1` is less than `ud2`. Both numbers are unsigned.

# 4.7 Floating point

There is a Floating Point package on disk in the file `FLOAT.SEQ` . To use Floating point in your programs type :

```
FLOAD  FLOAT.SEQ
```

The Floating Point words will then be loaded into the `FLOATING` vocabulary.

The format used for the floating point is the Motorola fast floating point format. This is a 32 bit format optimised for the 68000.

## FFP bit Format

MMMMMMMM MMMMMMMM MMMMMMMM SEEEEEEE

31 23 15 7 0

The meaning of the bits is as follows:

| | | |
|---|---|---|
| M | Mantissa | 24 bits |
| S | Sign of FFP | 1 bit |
| E | Exponent in excess-64 notation | 7 bits |

The mantissa is coded as a binary fixed-point fraction: it is normalised and represents a value of less than 1 but greater or equal to .5 .

The sign bit is set for a negative number and cleared for a positive number.

The exponent is a power of 2 used to raise the mantissa to its true value. It is in excess-64 notation, which means that 64 is added to it, so as it contains its own sign. It has a range of +63 to -64 and a zero exponent (E+0) would equal 64.

The range allowed for FFP is as follows:

+9.22337177 x 10**18 > +5.42101070 x 10**-20

-9.22337177 x 10**18 < -2.71050535 x 10**-20

## FFP Number Input

Numbers are input with a base 10 exponent. At the time of writing, all floating point numbers need to have an exponent, even if it is 0 (this would be `E+0`).

```
E+_    ( n --- ffp)
E+__   ( n --- ffp)
```

These words are used to input a positive exponent.

e.g.

```
1.0 E+1 1.0 E+10
```

```
E-_      ( n --- ffp)
E-__     ( n --- ffp)
```

These words are used to input a negative exponent.

e.g.

```
1.0 E-1 1.0 E-10
```

Here are some example floating point numbers:

```
1.25 E+4        -1.25 E+3        2.333 E-10        -100.45 E-0
```

`F+` `( ffp1 ffp2 --- ffp3)`
add `ffp1` to `ffp2` and leave the result `ffp3` on the stack.

`F-` `( ffp1 ffp2 --- ffp3)`
subtract `ffp2` from `ffp1` and leave the result `ffp3` on the stack.

`F*` `( ffp1 ffp2 --- ffp3)`
multiply `ffp1` by `ffp2` and leave the product `ffp3` on the stack.

`F/` `( ffp1 ffp2 --- ffp3)`
divide `ffp1` by `ffp2` and leave the result on the stack.

`F=` `( ffp1 ffp2 --- flag)`
`flag` is true if `ffp1` is equal to `ffp2`.

`F<` `( ffp1 ffp2 --- flag)`
`flag` is true if `ffp1` is less than `ffp2`.

`F>` `( ffp1 ffp2 --- flag)`
`flag` is true if `ffp1` is greater than `ffp2`.

`F0=` `( ffp --- flag)`
`flag` is true if `ffp` is zero.

`F0<` `( ffp --- flag)`
`flag` is true if `ffp` is less than zero.

`F<>` `( ffp1 ffp2 flag)`
`flag` is true if `ffp1` is not equal to `ffp2`.

`F<=` `( ffp1 ffp2 --- flag)`
`flag` is true if `ffp1` is equal to or greater than `ffp2`.

`F<=` `( ffp1 ffp2 --- flag)`
`flag` is true if `ffp1` is equal to or less than `ffp2`.

```
FABS      ( ffp --- +ffp)
```

Change `ffp` to its absolute value.

```
FNEGATE ( ffp --- ffp)
```

Reverse the sign of `ffp`.

```
F.        ( ffp)
```

The absolute value of `ffp` is displayed in a free field format with the exponent displayed as an E number. A leading negative sign is displayed if `ffp` is negative.

e.g.

```
ALSO  FLOATING              \ add  FLOATING  to  search  order
1.0  E+0  4.0  E+0  F/      \ divide  1  by  4
F.  1.250000E-1             \ print  result
F.R                ( ffp  n)
```

`ffp` is converted using the value of `BASE` and then displayed right aligned in a field `+n` characters wide with the exponent displayed as an E number. A leading negative sign is displayed if `d` is negative.

# 4.8 Assembler

The **HiSoft FORTH** Assembler is contained in the file `ASM.BLK`. It is derived from a 68000 Assembler written by Ken Mantel of California State College, San Bernadino and placed in the public domain.

Certain modifications have been made, in particular, it has been changed from 16 bit to 32 bit. Register names have been added and the opcode execution delayed so as to give a Motorola syntax.

## 4.8.1 Loading the Assembler

To load the Assembler while in the `SHELL` program, click on the `Open...` option from the `File` menu and then the `Loading...` option from the Screens menu.

To load the Assembler from the FORTH command line, type the following:

```
FLOAD  "ASM.SEQ"
```

The Assembler will now be loaded.

## 4.8.2 Using the Assembler

To use the Assembler with **HiSoft FORTH** you have 2 options:

1. CODE Words.

2. In-Line Assembly.

Here is an example of a CODE word, that puts a number on the stack

```
CODE NUM MOVE. .L 123 #W , -(A3) RTS. END-CODE
```

The equivalent using in-line assembly would be

```
: NUM { MOVE. .L 123 #W , -(A3) } ;
```

Both of the above would compile the same code and leave 123 on the Data Stack when executed.

As **HiSoft FORTH** uses subroutines and machine code macros for compiled code, there are no problems switching between FORTH and Assembly code.

## 4.8.3 Assembler Syntax

The table below documents differences between FORTH and Motorola Assembler syntax. As the FORTH assembler is made of FORTH words they need a space (or spaces) to separate individual words.

| Motorola | FORTH Assembler | |
|---|---|---|
| (A2) | (A2) | |
| (A2)+ | (A2)+ | |
| -(A2) | -(A2) | |
| 8 (A2) | 8 (A2) | |
| 20 (A2,D4.W) | 20 (A2,D4) | |
| -$11 (A2,A0.L) | -$11 (A2,A0) | |
| .LR $20 (PC) | $20 (PC) | (see note) |
| 8 (PC,D1.W) | 8 (PC,D1) | (see note) |
| 8 (PC,A0.L) | 8 (PC,A0) .LR | (see note) |
| $FA00 (absolute short) | $FA00 ABSW | |
| $FA00FF00 (absolute long) | $FA00FF00 ABSL | |
| #27 (immediate byte or word) | 27 #W | |
| #$ABCD00 (immediate long) | $ABCD00 #L | |
| CCR | CCR | |
| SR | SR | |
| USP | USP | |
| D0/D3-D6/A2-A3 | D0-D0 D3-D6 A2-A3 (MOVEM.) | |

The Assembler also recognises `(SP)`, `(SP)+` and `-(SP)` . `4(SP)` should be expressed as `4 &(SP)` .

Note: The PC-relative instructions require that an absolute address is provided as the parameter. The Addressing Mode words take this address and turn it into an offset by deducting the value of `HERE` at the point of execution. The resultant offset must be in the range -32,768 to 32,767.

All opcodes include a . period at the end of the mnemonic. So `MOVE` becomes `MOVE.` .

Source and Destination operands and separated by a , (comma) . The FORTH word , is redefined for use in the Assembler as `D,` or `A,` for addresses.

Here are some examples of FORTH Assembler in use

| **Motorola** | **FORTH** | |
|---|---|---|
| `move.l d0,(a2)+` | `MOVE. .L D0 , (A2)+` | |
| `cmpi.b #65,d0` | `CMP. .B 65 #W , D0` | |
| `move d0,-(a3)` | `MOVE. .W D0 , -(A3)` | (.W optional) |
| `asl #2,d1` | `ASL.2 #L , D1` | |
| `movem.l d2-d4,-(sp)` | `MOVEM. .L D2-D4 , -(SP)` | |

Notice that the syntax is similar, but that the FORTH must be typed in upper-case and spaces must be left between FORTH words.

## 4.8.4 Using HiSoft FORTH with DevpacST

While the FORTH Assembler is fine for most uses, for larger assembler programs you may prefer to use **HiSoft DevpacST**. The Assembler `.PRG` files can then be loaded by FORTH and combined for stand-alone programs using the `.PRG` compiler.

To load the assembler file into memory you could use the following FORTH code

```
ALSO TOS

VARIABLE BASE-ADDR

: BLOAD ( addr)
     1+ " " 1+ DUP 3 EXEC DUP 0< 7 ?ERROR
     256 + BASE-ADDR ! ;

: HEADER ( n)
  <BUILDS
    4* BASE-ADDR @ + ,
    DOES> @ EXECUTE ;

PREVIOUS

" MYFILE.PRG" BLOAD

1 HEADER CONIN
2 HEADER CONOUT
```

This would load an assembler program, for example the following one, with two GEMDOS routines, the assembler code would be

```
myfile: bra     exit                ; optional exit or startup
        dc.l    conin-*             ; pointer to conin
        dc.l    conout-*            ; pointer to conout
                                    ; console output routine
conin:  move.w  #1,-(sp)
        trap    #1
        addq.l  #2,sp
        move.l  d0,-(a3)            ; leave character on stack
        rts                         ; console input routine

conout: move.l  (a3)+,d0            ; get character from stack
        move.w  d0,-(sp)
        move.w  #2,-(sp)
        trap    #1
        addq.l  #4,sp
exit:   rts
```

The word HEADER is used to associate the table at the start of the file with a given FORTH word. Thus

```
1 HEADER CONIN
```

uses the first entry in the table.

If you use this method make sure to use SHRINK to return enough memory to the system.

# 4.9 Multi-Tasking

**HiSoft FORTH** supports multi-tasking using a round-robin loop multi-tasker. This type of multi-tasker is controlled by the user and uses the word PAUSE to switch between tasks. A task in this case is a FORTH word that can run in the background, while you are using the computer for other things.

These words are in the TASK vocabulary.

`USER ( n --- addr)`

Define a task user variable. Similar to SYS_VAR (system variables). A USER variable is switched with each task. Each task would have its own copy of a user variable.

`PAUSE ( --- )`

Switch between tasks in round-robin loop.

`LOCAL ( addr task --- addr1)`

Allow access to a USER variable in another task. e.g.

```
BASE TASK1 LOCAL
```

would return the address of the BASE user variable in task TASK1.

`ACTIVATE ( task)`

Wake task and make it execute following code. e.g.

```
TASK: +CNTO VARIABLE CNT
: COUNTER
        +CNT ACTIVATE
          BEGIN 1 CNT +! PAUSE AGAIN STOP ;
```

`COUNTER` would start task `+CNT` incrementing a count in variable `CNT`.

`SLEEP ( task)`

Put task to sleep.

`WAKE ( task)`

Set task to wake up on next pass of round-robin loop.

`STOP ( --- )`

Stop current task.

`MULTI ( --- )`

Set multi-tasking on.

`SINGLE ( --- )`

set multi-tasking off.

`TASK: ( --- )`

A task defining word, see `ACTIVATE` for an example of usage.

`V: ( n)`

define vector handler in FORTH. `n` is an exception vector number. This is used in place of colon `:`.

e.g.

```
        36 V: FOUR CR ." Hello World!!!" CR ;V
```

would set up TRAP #4 to execute print statement. 36 is used because the `TRAP #` instruction vectors are traps 32 to 47.

To test it type

```
        4 #TRAP
```

`;V ( --- )`

used to end vector definition.

`#TRAP ( n)`

executes trap number `n`.

# 4.10 MIDI library

The MIDI library provides a few simple words that are slightly easier to use than the 'raw' operating calls. It is loaded using

```
FLOAD  MIDI.SEQ
```

and contains the following words:

```
>M      ( 8b --- )
```

Outputs a single byte 8b to the MIDI port.

```
M>      ( --- 8b )
```

Reads a single byte from the MIDI port.

```
?MI     ( --- f )
```

Returns a non-zero value if a character is waiting to be input from the MIDI port.

```
?MO     ( --- f)
```

Returns a non-zero value if a character may be output to the MIDI port.

```
KEY.ON  ( note --- )
```

Switches on the given `note`.

```
KEY.OFF ( note --- )
```

Switches off the given `note`.

```
KEYS.ON ( note1...noten n --- )
```

Switches on a range of notes. Just before calling this word you should place the number of notes on the stack.

e.g.

```
12  13  14  3  KEYS.ON
```

switches the 3 notes 12, 13 and 14 on.

```
KEYS.OFF     ( note1...note.n n --- )
```

Switches off a range of notes; this word works in a similar way to `KEYS.ON`.

```
KCLR   ( --- )
```

Clears all notes.

```
?MIDI  ( --- )
```

Echoes bytes received from the MIDI port until a key on the computer's keyboard is pressed.

# 4.11 Graphics

**HiSoft FORTH** supports Atari graphics using line-A calls. There is also a set of Turtle graphics words for drawing shapes. These words are in the built-in `GRAPHIC` vocabulary.

`FD ( n)` 'forwards'

move forwards n pixels.

`BK ( n)` 'backwards'

move backwards n pixels.

`RT ( angle)` 'right-turn'

turn right by angle.

`LT ( angle)` 'left-turn'

turn left by angle.

`PD ( --- )` 'pen-down'

draw line when turtle moves.

`PU ( --- )` 'pen-up'

don't draw line when turtle moves.

`PEN ( --- addr )`

Variable used to indicate PEN state.

`MVTO ( x y)` 'move-to'

move to location (x,y) without drawing line.

`HEAD ( angle)`

set current heading to angle.

`TURTLE ( --- x y head)`

Return current x,y location and current heading.

`A-LINE ( - addr)`

Returns the address of the Line-A variables.

`PIXEL ( x y --- c)`

Return pixel colour c at location x,y.

`PLOT ( x y c)`

Plot point on screen at x,y location in colour c.

`DRAW ( x y mode)`

Draw a line to x,y from current x,y location, use drawing mode.

`COLOR ( n)`

Set colour value for PLOT.

`PLANES ( n)`

Set drawing planes used by `DRAW`, `RECT` and `POLY`. `n` should be a number between 0-15 .

`RECT ( x1 y1 x2 y2)`

Draw a filled rectangle with `x1`, `y1` as top left corner and `x2`, `y2` as bottom right corner of rectangle.

`POLY ( addr n)`

Draw a filled in polygon using co-ordinates in an word array at `addr` with `n` points. The first point must be repeated as the last point to complete the polygon.

`CLIP ( x1 y1 x2 y2 mode)`

define clipping rectangle with `x1`, `y1` as top left corner and `x2`,`y2` as bottom right corner. `mode` = 0 for no clip, `mode` = 1 for clipping.

`SPRITE ( x y sprite buffer)`

draw a sprite at `x`,`y` location saving the screen to `buffer`.

`XSPRITE ( buffer)`

Un-draw sprite by restoring screen from `buffer`.

`COS ( angle --- cos)`

Return COSINE `cos` of angle from look-up table.

`SINE ( angle --- sine)`

Return SINE `sine` of angle from look-up table.

## 4.12 Atari ST Extensions

These words are included to access ATARI routines.

`WRAP ( --- )`

Set text to wrap at end of line.

`UNWRAP ( --- )`

Set text to fixed line.

`CSRON ( --- )`

Turn cursor on.

`CSROFF ( --- )`

Turn cursor off.

`MON ( --- )`

Turn mouse cursor on.

`MOFF ( --- )`

Turn mouse cursor off.

`HOME ( --- )`

Home cursor to top left hand corner of screen.

`FGND ( n)`

Set foreground colour to `n`. The colours set depend on the graphics mode and the colour palette.

`BKGND ( n)`

Set background colour to `n`.

# 5 Direct Operating System Calls

## 5.1 GEMDOS

TOS (The Operating System) is the operating system used by the Atari ST. It is made up of different levels.

The hardware level is the BIOS and XBIOS (Basic Input Output System and eXtended BIOS). This allows calls to the hardware plus disk sector and formatting calls. The next level up is hardware independent level called GEMDOS. This has file handling calls and gives access to low level operating system calls.

GEMDOS was developed by Digital Research as a new operating system for 68000 computers; most of the calls emulate those on MS-DOS computers. There are a large number of calls that are implemented as high level FORTH words. Below are short descriptions and also the stack parameters before and after the calls. These should be used in conjunction with more detailed TOS and GEM documentation. See the **Bibliography**.

```
TERM ( --- )
```

Returns to the program from which it was started.

```
PTERMRES ( size code --- )
```

Terminate with a return code, but keep the program's code in memory. size indicates how much memory from the program start should remain allocated. code is the return code.

```
PEXEC ( runflag pathname tail environ --- result )
```

Load a program from disk. `runflag` indicates `0` = run, `3` = load only. `pathname` is a pointer to a pathname of the file. `tail` points to a command tail for the program. `environ` is a pointer to its environment strings. If the file was loaded the result is the load address. If the file was run the result is the return code.

```
PRETURN ( code --- )
```

Terminate returning a code.

```
CCONIN ( --- char )
```

Read a character char from the console.

```
CCONOUT ( char --- )
```

Write a character char to the console.

```
CAUXIN ( --- char )
```

Read a character from the auxiliary device (modem port).

`CAUXOUT ( char --- )`

Write a character `char` to the auxiliary device (modem or serial port).

`CPRNOUT ( char --- )`

Write a character `char` to the printer.

`CRAWIO ( char --- (char) )`

If `char` is not $FF (255) then write it as a character to the console, otherwise return a character (`char`) from the console with no echo.

`CRAWCIN ( ---- char )`

Read a character from the console with no echo or control character trapping.

`CNECIN ( --- char )`

Read character with no echo, but trap `^C ^S` and `^Q`.

`CCONWS ( string --- )`

Write zero terminated string to console

`CCONRS ( buffer --- )`

Read a line of characters allow line edit

`CCONIS ( --- status )`

Check the status of the console input device. Returns -1 (true) if character waiting, else 0 if none available.

`CCONOS ( --- status )`

Check the status of the console output device. Returns -1 (true) if character waiting, else 0 if none available.

`CPRNOS ( --- status )`

Check the status of the printer. Returns -1 (true) if character waiting, else 0 if none available.

`CAUXIS ( --- status )`

Check the status of the auxiliary input device. Returns -1 (true) if a character waiting, else 0 if none available.

`CAUXOS ( --- status )`

Check the status of the auxiliary output device. Returns -1 (true) if a character waiting, else 0 if none available.

`TGETDATE ( --- date )`

Returns the current system date on the stack. Bits 0-4 of the result contain the date, 5-8 contain the month, 9-15 contain the year minus 1980 (up to 2099).

`TSETDATE ( date --- )`

Set the current system date to date on the stack.

```
TGETTIME ( --- time )
```

Return the current system time on the stack. Bits 0-4 contain seconds/2, 5-10 contain minutes, 11-15 contain hours.

```
TSETTIME ( time --- )
```

Set the current system time to time on the stack.

```
SUPER ( 0 --- SSP )
      ( SSP --- )
```

When this function is called with a value of zero it returns the supervisor stack pointer (`SSP`), which should be saved, it also places the 68000 processor in supervisor mode. **HiSoft FORTH** and most other programs usually run in the user mode of the processor. The supervisor mode is used by the operating system. In supervisor mode you have full access to the machine's memory, but you need to be careful. To return to the user mode use the old `SSP` as a parameter to `SUPER`.

```
VERSION ( --- version )
```

Calling this function returns the version number of GEMDOS.

```
DSETDRV ( drive --- )
```

Set the default disk to drive. Values 0-15 indicate drives A-P.

```
DGETDRV ( --- drive )
```

Return the value of current drive.

```
DFREE ( buffer drive --- )
```

Get information about `drive`. `buffer` is the address of a buffer to receive the information. `drive` indicates the drive to get the information from. The buffer is 16 bytes long. It gets 4 values, free space, Total clusters, size of sector in bytes and size of cluster in sectors.

```
DMKDIR ( pathname --- )
```

Create a subdirectory. `pathname` is the addr of a null (zero) terminated string for the pathname of the new directory.

```
DRMDIR ( pathname --- )
```

Remove a subdirectory.

```
DCHDIR ( pathname --- )
```

Change to a different subdirectory.

```
DGETDIR ( buffer drive --- )
```

Store the current directory in a 64 byte buffer pointed to by buffer. drive is the drive to search 0 = current, 1 = drive A, 2 = drive B, etc.

```
FSETDTA ( DTAbuffer --- )
```

Set disk transfer address (DTA). The DTA is the address of a 44 byte buffer used when searching for a file.

`FGETDTA ( --- DTAbuffer )`

Return the address of the current DTA buffer.

`FCREATE ( pathname attributes --- handle )`

Create a file named by the null terminated string pointed to by the address pathname. The `attributes` are as follows

| | |
|---|---|
| 0 | Normal file status, read/write |
| 1 | Read only file |
| 2 | Hidden file |
| 4 | System file |
| 8 | Volume label, contains disk name |
| 16 | Subdirectory |
| 32 | File is written and closed |

A file `handle` is returned on the stack. At the time you `FCREATE` a file, you can use the file handle without opening the file. A total of 40 files can be open at the same time.

`FOPEN ( pathname access --- handle )`

Open a file named by the null terminated string pointed to by the address pathname. The access modes are as follows

| | |
|---|---|
| 0 | Read only |
| 1 | Write only |
| 2 | Read and write |

The function returns a file handle if the access mode is possible, otherwise it returns an error code. see GEMDOS error codes.

`FCLOSE ( handle --- )`

Close a file that has been opened `FOPEN` given the `handle` on the stack.

`FREAD ( handle count buffer --- return )`

Read from a file. `handle` is an open file handle. `count` is the number of bytes to transfer. `buffer` is the address of a buffer to which the file is to be read. `return` is the number of bytes read or a GEMDOS error number.

`FWRITE ( handle count buffer --- return )`

Write to a file. `handle` is an open file handle. `count` is the number of bytes to transfer. `buffer` is the address of a buffer which the file is to write to. `return` is the number of bytes read or a GEMDOS error number.

`FDELETE ( pathname --- )`

Delete a file. `pathname` is the address of a null terminated string pathname of the file.

`FSEEK ( count handle mode --- position )`

Move the file pointer. `count` is a byte count pointer. `handle` is file handle of an open file. mode is as follows

| | |
|---|---|
| 0 | count forwards from start of file |
| 1 | relative count form current position |
| 2 | count backwards from end of file |

`position` is the actual position set from the beginning of the file.

`FATTRIB ( pathname mode attribute --- )`

Read or change the file attributes. `pathname` is a pointer to a null terminated string pathname. `mode` is 0 = get, 1 = set. for attributes .See `FCREATE`.

`FFORCE ( handle1 handle2 --- )`

Force `handle1` to point to the same file as `handle2`.

`FSFIRST ( pathname attribute --- return )`

Search for the first file which matches the search string pointed to by `pathname`. The string can contain the wildcards `*` or `?`. For search file attributes see `FCREATE`. Before using this call you set up a DTA buffer (see `FSETDTA`),which this call will return the file size and file name of the file found. `return` contains zero if the file is found or the GEMDOS error -33 file not found.

`FSNEXT ( --- return )`

Use this call after `FSFIRST` to find another match of the search string.

`FRENAME ( oldname newname --- )`

Rename a file. `oldname` is pointer to filename to be renamed. `newname` is pointer of new file name to be used.

`FDATIME ( buffer handle mode --- )`

Get or set a files date or time. `buffer` is a pointer to a two word (4 byte) buffer (a time word and a date word). `handle` is the file handle of an open file. `mode` is 0 = set, 1 = get.

`MALLOC ( count --- addr )`

Allocate memory block. `count` is number of bytes to allocate and `addr` is start of memory block returned by system or an error if negative. If `count` is set to -1 then the system will return maximum free memory available.

`MFREE ( addr --- error )`

Used after `MALLOC` to return memory block back to system. addr is a previous address obtained from `MALLOC`. If `error` = 0 then memory was released ok; a negative value indicates an error.

# 5.2 The BIOS

The BIOS ( Basic Input/Output System) is the interface between GEMDOS and the Hardware of the ST.

`GETMBP ( p_mpb --- )`

On entry addr `p_mpb` points to a 12 byte block of memory to be filled in with the system initial Memory Parameter Block. On return the block is filled in with three pointers as follows

```
MPB

MD_addr1  *mp_mfl      memory free list
MD_addr2  *mp_mal      memory allocated list
MD_addr3  *mp_rover    roving pointer
```

each pointer points to a structure as follows

```
MD

MD_addr     *m_link      next MD or NULL
addr        m_start      start address of block
addr        m_length     #bytes in block
PD_addr     *m_own       owner's process descriptor
```

`BCONSTAT ( dev --- flag )`

Return character device input status, `flag` will be false if no characters available or true if at least one character is available. `dev` can be one of

```
0       PRT:      printer, the parallel port
1       AUX:      aux device, the RS-232 port
2       CON:      console, the screen
3       MIDI      midi port
4       IKBD      keyboard port
```

`BCONIN ( dev --- char )`

Does not return until a character has been input ( busy wait). It returns the character on the stack. For `CON:` ( `dev=2` ) it returns a scancode in the lower byte of the upper word.

`BCONOUT ( dev c --- )`

Output character `c` to the device `dev`. Does not return until the character has been written.

`RWABS ( rwflag buf count recno dev --- error_code )`

Read or write logical sectors on a device. `rwflag` is one of

| | |
|---|---|
| 0 | read |
| 1 | write |
| 2 | read, do not affect media-change |
| 3 | write, do not affect media-change |

`buf` points to a buffer to read or write. `count` is the number of sectors to transfer. `recno` is the logical sector number to start the transfer at. `dev` is the device number. On the ST this is one of

| | |
|---|---|
| 0 | Floppy drive A: |
| 1 | Floppy drive B: |
| 2+ | Hard disks, Networks, etc. |

`SETEXEC ( vecnum vec --- vec1 )`

`vecnum` is the number of the vector to get or set. `vec` is the address to set up in the vector slot. If `vec` is -1 then `vec1` is value of vector set. When setting a vector with this call you should drop the value on the stack. `vec1` is the previous value of the vector when setting a vector.

`GETBPB ( dev --- bpb_addr )`

`dev` is a device number (0 = drive A: ,etc ). Returns a pointer to the BIOS parameter block for the specified drive. An address of zero a bit position (0..31) when a drive is available for that bit, or a 0 if not.

`KBSHIFT ( mode --- return )`

Determines the status of special keys on the keyboard. If `mode` is -1 you get the status, a positive value is accepted as the new status. The status is a bit vector which is as follows

| **Bit** | **Meaning** |
|---|---|
| 0 | Right `shift` key |
| 1 | Left `shift` key |
| 2 | `Control` key |
| 3 | `ALT` key |
| 4 | Caps Lock on |
| 5 | `Clr Home` |
| 6 | `Insert` |
| 7 | Unused |

## 5.3 The Extended BIOS (XBIOS)

```
INITMOUSE   ( type param vec ---- )
```

Initialise mouse.

```
SSBRK ( amount --- )
```

Save memory space.

```
PHYSBASE    ( --- addr )
```

`addr` is the base of physical screen ram.

```
LOGBASE ( --- addr )
```

`addr` is the logical screen base used as the screen base for screen output.

```
GETREZ  ( --- n )
```

`n` is the screen resolution.

| | |
|---|---|
| 0 = low resolution | 320 x 200 |
| 1 = medium resolution | 640 x 200 |
| 2 = high resolution | 640 x 400 |

```
SETSCREEN   ( logloc physloc rez --- )
```

Change the screen parameters for logical base, physical base and resolution. If a parameter is to be left unchanged a -1 value should be passed on the stack.

```
SETPALLETE  ( paletteptr --- )
```

Load new colour palette. `paletteptr` is a pointer to a table of 16 colours (each colour takes 16 bits). The colours will be loaded at the next VBL interrupt.

```
SETCOLOR    ( colornum colour --- )
```

Change one colour. `colornum` is the colour number (0-15) and `colour` is the colour (0-$777) to set.

```
FLOPRD  ( buf filler devno secno trackno sideno
          count --- status )
```

Read one or more sectors from the disk.

```
FLOPWR  ( buf filler devno secno trackno sideno
          count ---   status )
```

Write one or more sectors to the disk.

```
FLOPFMT ( buf filler devno spt trackno sideno
        interlev magic virgin --- status )
```

Format a disk track.

```
MIDIWS  ( count ptr --- )
```

Output a string to the MIDI port. `ptr` points to the string and `count` contains the number of characters to send-1.

```
MFPINT  ( intno vector --- )
```

Initialise an interrupt routine in the MFP 68901. `intno` is the interrupt number, vector is the interrupt routine address.

```
IOREC ( --- devno )
RSCONF  ( speed flowctl ucr rsr tsr scr --- )
```

Configure the RS-232 port.

```
KEYTBL  ( unshift shift capslock --- keytab_addr )
```

Set keyboard table.

```
RANDOM  ( --- 24b )
```

Return a 24 bit random number.

```
PROTOBT ( buf serialno disktype execflag --- )
```

Produce boot sector.

```
FLOPVER ( buf filler devno secno trackno sideno
        count ---   status )
```

Verify one or more sectors on a disk.

```
SCRDMP  ( --- )
```

Output a hardcopy of the screen to the selected printer.

```
CURSCONF    ( rate attrib --- status )
```

Configure the cursor.

```
SETTIME ( datetime --- )
```

Set clock time and date.

```
GETTIME ( --- datetime )
```

Return time and date.

```
BIOSKEYS    ( --- )
```

Restore BIOS keyboard table.

```
IKBDWS  ( cnt ptr --- )
```

Send commands to intelligent keyboard processor.

```
JDISINT ( intno --- )
```

Selectively disable interrupts on the MFP68901. `intno` is the interrupt number (0-15).

```
JENABINT    ( intno --- )
```

Re-enable interrupts disabled by `JDISINT`.

```
GIACCESS    ( data access --- )
```
Access the registers on the GI sound chip.

```
OFFGIBIT    ( bitno --- )
```
Set a bit of Port A of the sound chip.

```
ONGIBIT ( bitno --- )
```
Clear a bit of Port A of the sound chip.

```
XBTIMER ( timer control data vec --- )
```
Start MFP 68901 timer.

```
DOSOUND ( ptr --- )
```
Set sound parameters.

```
SETPRT  ( config --- )
```
Configure printer.

```
KBDVBASE    ( --- kbdvecs_addr )
```
Return keyboard vector table.

```
KBRATE  ( initial repeat --- )
```
Set keyboard repeat rate.

```
VSYNC ( --- )
```
Wait for video.

```
PRTBLK  ( addr --- )
```
Output block to printer.

```
SUPEREXEC   ( addr --- )
```
Set supervisor execution.

```
PUNTAES ( --- )
```
Disable GEM AES.

# 5.4 GEM VDI (Virtual Device Interface)

The GEM VDI is called from FORTH by placing parameters on the FORTH data stack. The parameters are in the same order as for C or Assembler. They also use the Digital Research names for the different functions.

e.g. The C binding for `v_gtext` is

```
v_gtext(handle,x,y,string)
```

in FORTH this would be

```
V_GTEXT ( handle x y string --- )
```

The above FORTH word would expect 4 values on the data stack the first the file handle, the second and third the `x` and `y` values and the top of stack would be the address pointer to the string.

Here's the state, of the data stack before and after a `V_GTEXT` call

| | before | after |
|---|---|---|
| TOS | String | empty stack |
| NOS | x | |
| 3rd | y | |
| 4th | handle | |
| | empty stack | |

Where it is not practical to pass all the required parameters via the data stack, it is noted under the GEM word stack picture. In this case the GEM arrays must be filled before the call with any extra values or addresses as needed.

## 5.4.1 GEM VDI arrays

The following arrays are pre-defined arrays used by the GEM VDI.

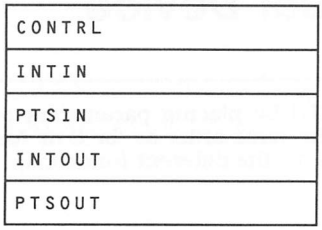
They expect a cell number on the stack and return an address that can used like a FORTH variable. If the array is a word array then use `W!` and `W@` to store and fetch word values between the array and the stack.

If the array is a long-word array then use `!` and `@` to fetch long-words or 32 bit addresses between the array and the stack.

`PB ( n --- addr )` address array

This is the GEM VDI Parameter block array. It is pre-defined with the addresses of the other GEM arrays.

PB array




```
CONTROL ( n --- addr )  word array
INTIN   ( n --- addr )  word array
PTSIN   ( n --- addr )  word array
INTOUT  ( n --- addr )  word array
PTSOUT  ( n --- addr )  word array
VDISYS  ( --- )
```

call GEM VDI. The VDI arrays must be set up before this call.

```
>VDI   ( PB_addr --- )
```

This word is the used to pass preset AES arrays to GEM. Otherwise it is similar to `VDISYS`.

## 5.4.2 GEM VDI Control Functions

```
V_OPNWK ( --- device_handle )
```

The Open Workstation call loads a graphics driver for the application and returns a device handle. The device is initialised with the parameters from the `WORKIN` array. Information about the device is returned in the `WORKOUT` array.

Output Parameters:

`device_handle` +n = device handle, 0 = device can not be opened

`WORKIN` parameters:

| | |
|---|---|
| 0 | Device ID number |
| 1 | Line type |
| 2 | Line colour index |
| 3 | Marker type |
| 4 | Marker colour index |
| 5 | Text face |
| 6 | Text colour index |
| 7 | Fill interior style |

| | |
|---|---|
| 8 | Fill style index |
| 9 | Fill colour index |
| 10 | NDC to RC transformation flag<br>0 = Map full NDC to RC<br>1 = Reserved<br>2 = Use the RC system |

WORKOUT Parameters:

| | |
|---|---|
| 0 | Maximum width of screen in rasters |
| 1 | Maximum Height of screen in rasters |
| 2 | Device coordinate units flag<br>0 = capable of precisely scaled image<br>1 = not capable of precisely scaled image |
| 3 | Width of one pixel in microns |
| 4 | Height of one pixel in microns |
| 5 | Number of character heights<br>0 = continuous scaling |
| 6 | Number of line types |
| 7 | Number of line widths<br>0 = continuous scaling |
| 8 | Number of marker types |
| 9 | Number of marker sizes<br>0 = continuous scaling |
| 10 | Number of faces supported |
| 11 | Number of patterns |
| 12 | Number of hatch styles |
| 13 | Number of predefined colours |
| 14 | Number of Generalised Drawing Primitives (GDPs) |
| 15 to 24 | Linear list of the first 10 supported GDPs<br>The number indicates which GDP. -1 indicates the end of the list. GEM VDI defines 10 GDPs<br>1 Bar<br>2 Arc<br>3 Pie slice<br>4 Circle<br>5 Ellipse<br>6 Elliptical arc<br>7 Elliptical pie<br>8 Rounded rectangle<br>9 Filled rounded rectangle<br>10 Justified graphics text |

| | |
|---|---|
| 25 to 34 | Linear list of attribute set with each GDP<br>0 Polyline<br>1 Polymarker<br>2 Text<br>3 Fill area<br>4 None |
| 35 | Colour capability flag<br>0 no<br>1 yes |
| 36 | Text rotation capability flag<br>0 no<br>1 yes |
| 37 | Fill area capability flag<br>0 no<br>1 yes |
| 38 | Cell array operation capability flag<br>0 no<br>1 yes |
| 39 | Number of available colours<br>0 continuous device ( >32767 colours)<br>2 monochrome<br>>2 number of colours |
| 40 | Number of locator devices<br>1 Keyboard only<br>2 Keyboard and other input |
| 41 | Number of valuator devices<br>1 Keyboard only<br>2 if another valuator device is available |
| 42 | Number of choice devices<br>1 function keys on keyboard<br>2 if another keypad is available |
| 43 | Number of string devices<br>1 keyboard |
| 44 | Workstation type<br>0 output only<br>1 input only<br>2 input/output<br>4 metafile output |

`V_CLSWK ( device_handle --- )`

The Close Workstation call terminates the graphic device. If the device is a printer an update occurs. For a metafile, GEM VDI flushes the buffer and close the file.

Input Parameters: `device_handle`

Output Parameters: NONE

`V_OPNVWK ( handle1 --- handle2 )`

Allows a single physical device to work as multiple workstations. Each virtual workstation has access to the whole screen, but the attributes are set separately.

The input the Open Virtual Workstation is a device handle of an open physical screen, This can be obtained from the `GRAF_HANDLE` call. The `WORKIN` array and the `WORKOUT` array are set as in the `V_OPNWK` call.

Input Parameters:

`handle1` device handle of an open physical screen

Output Parameters:

`handle2` `+n` = device handle, `0` = cannot open device

`V_CLSVWK ( device_handle --- )`

Terminates virtual device and prevents further screen output to device.

Input Parameters:

`device_handle` virtual device handle

Output Parameters: NONE

`V_CLRWK ( device_handle --- )`

Clear Workstation. Erase screen, form feed printer or output opcode to metafile.

Input Parameters: `device_handle`

Output Parameters: NONE

`V_UPDWK ( device_handle --- )`

Execute all pending graphic commands. For printer drivers you must use this function to start output. For a metafile, GEM VDI outputs the opcodes.

Input Parameters: `device_handle`

Output Parameters: NONE

`VS_CLIP ( handle clip_flag x1 y1 x2 y2 --- )`

Enable or disable clipping of GEM VDI. The default is for clipping disabled.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| clip_flag | clipping flag<br>0 = turn clipping off<br>+n = turn clipping on |
| x1 | x-coord of upper right corner to clip |
| y1 | y-coord of upper right corner to clip |
| x2 | x-coord of lower left corner to clip |
| y2 | y-coord of lower left corner to clip |

Output Parameters:NONE

## 5.4.3 GEM VDI Output Functions

`V_PLINE ( handle count --- )`

Display a 'poly line' on a graphics device. GEM VDI will not display a single coordinate line. Lines are drawn using the current line attributes:

- colour
- line type
- line width
- end style
- current writing mode

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `count` | number of vertices i.e. (x,y pairs) to be drawn |

`pxarray` array of coordinates of poly line in the `PTSIN` word array. This array must be filled before using this function.

| | |
|---|---|
| `0 PTSIN` | x-coord of 1st point |
| `1 PTSIN` | y-coord of 1st point |
| `2n-2 PTSIN` | x-coord of last point |
| `2n-1 PTSIN` | y-coord of last point |

Output Parameters: NONE

`V_PMARKER ( handle count --- )`

Draws markers at the points specified in the `PTSIN` input array. GEM VDI displays the markers using the current marker attributes:

- colour
- scale
- type
- writing mode

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `count` | number of vertices i.e. (x,y pairs) to be drawn |

`pxarray` array of coordinates for the markers in `PTSIN` word array. This array must be filled before using this function.

| | |
|---|---|
| `0 PTSIN` | x-coord of 1st point |
| `1 PTSIN` | y-coord of 1st point |
| `2n-2 PTSIN` | x-coord of last point |
| `2n-1 PTSIN` | y-coord of last point |

Output Parameters: NONE

`V_GTEXT ( handle x y string --- )`

Display graphic text at the `x, y` alignment point. The default alignment is the left baseline of the text string or it can be changed with `VST_ALIGNMENT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | x-coord of alignment point of text |
| `y` | y-coord of alignment point |
| `string` | address of null terminated string |

Output Parameters: NONE

`V_FILLAREA ( handle count --- )`

Fills a complex polygon specified in `PTSIN` input array. The area is filled using the current attributes:

- fill area colour
- interior style
- writing mode
- style

Input Parameters:

| | |
|---|---|
| handle | device handle |
| count | number of vertices i.e. (x,y pairs) to be drawn |

pxarray array of coordinates for the the area to be filled in PTSIN word array. This array must be filled before using this function.

| | |
|---|---|
| 0 PTSIN | x-coord of 1st point |
| 1 PTSIN | y-coord of 1st point |
| 2n-2 PTSIN | x-coord of last point |
| 2n-1 PTSIN | y-coord of last point |

Output Parameters: NONE

```
V_CELLARRAY ( handle x1 y1 x2 y2 row_length el_used
      num_rows wrt_mode --- )
```

Draw a rectangular array defined by the x,y coordinates and the colour index array in INTIN. GEM VDI divides the rectangle into cells based on the number of rows and columns and the colour index array specifies the colour for each cell.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x1 | x-coord of lower right corner |
| y1 | y-coord of lower left corner |
| x2 | x-coord of upper right corner |
| y2 | y-coord of upper right corner |
| row_length | length of row in colour index array |
| el_used | number of elements in each row |
| num_rows | number of rows in colour index array |
| wrt_mode | pixel operation |

Output Parameters:NONE

`V_CONTOURFILL ( handle x y index --- )`

Flood fill area until edge or colour index. If index is negative, the algorithm searches for any colour other than the seed colour.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | x-coord of starting point |
| `y` | y-coord of starting point |
| `index` | colour index |

Output Parameters: NONE

`VR_RECFL ( handle x1 y1 x2 y2 --- )`

Fills rectangular area with pattern defined by current fill area attributes.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x1` | x-coord of upper right corner |
| `y1` | y-coord of upper right corner |
| `x2` | x-coord of lower left corner |
| `y2` | y-coord of lower left corner |

Output Parameters: NONE

`V_BAR ( handle x1 y1 x2 y2 --- )`

Draw a filled bar.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x1` | x-coord of upper right corner |
| `y1` | y-coord of upper right corner |
| `x2` | x-coord of lower left corner |
| `y2` | y-coord of lower left corner |

Output Parameters: NONE

`V_ARC ( handle x y radius begang endang --- )`

Draw an arc.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | x-coord of centre point |
| `y` | y-coord of centre point |
| `radius` | radius |
| `begang` | start angle (0-3600) |
| `endang` | end angle (0-3600) |

Output Parameters: NONE

`V_PIE ( handle x y radius begang endang --- )`

Draw a pie slice.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | x-coord of centre point |
| `y` | y-coord of centre point |
| `radius` | radius |
| `begang` | start angle (0-3600) |
| `endang` | end angle (0-3600) |

Output Parameters: NONE

`V_CIRCLE ( handle x y radius --- )`

Draw a circle.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | x-coord of centre point |
| `y` | y-coord of centre point |
| `radius` | radius |

Output Parameters: NONE

```
V_ELLIPSE ( handle x y xradius yradius --- )
```

Draw an ellipse.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x1 | x-coord of centre point |
| y1 | y-coord of centre point |
| xradius | radius of x-axis |
| yradius | radius of y-axis |

Output Parameters: NONE

```
V_ELLARC ( handle x y xradius yradius begang endang
        --- )
```

Draw an elliptical arc.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x | x-coord of centre point |
| y | y-coord of centre point |
| xradius | radius of x-axis |
| yradius | radius of y-axis |
| begang | start angle (0-3600) |
| endang | end angle (0-3600) |

Output Parameters: NONE

```
V_ELLPIE ( handle x y xradius yradius begang endang
        --- )
```

Draw an elliptical pie slice.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x | x-coord of centre point |
| y | y-coord of centre point |
| xradius | radius of x-axis |
| yradius | radius of y-axis |
| begang | start angle (0-3600) |
| endang | end angle (0-3600) |

Output Parameters: NONE

```
V_RBOX  ( handle x1 y1 x2 y2 --- )
```
Draw a rectangle with rounded corners

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x1 | x-coord of upper right corner |
| y1 | y-coord of upper right corner |
| x2 | x-coord of lower left corner |
| y2 | y-coord of lower left corner |

Output Parameters: NONE

```
V_RFBOX ( handle x1 y1 x2 y2 --- )
```
Draw a filled rectangle with rounded corners.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x1 | x-coord of upper right corner |
| y1 | y-coord of upper right corner |
| x2 | x-coord of lower left corner |
| y2 | y-coord of lower left corner |

Output Parameters: NONE

```
V_JUSTIFIED ( handle x y string length word_space
      char_space  --- )
```
Output left and right justified graphics text to the workstation.

Input parameters:

| | |
|---|---|
| handle | device handle |
| x | x-coord of alignment point of text |
| y | y-coord of alignment point |
| string | string of text |
| length | length of text in x-axis units |
| word_space | inter-word spacing<br>0 = do not modify spacing<br>non-zero = allows GEMVDI to modify spacing |
| char_space | inter-word spacing<br>0 = do not modify spacing<br>non-zero = allows GEMVDI to modify spacing |

Output Parameters: NONE

## 5.4.4 GEM VDI Attribute Functions

`VSWR_MODE ( handle mode --- set_mode )`

Select writing mode used for drawing operations.

Writing Modes:

| | |
|---|---|
| 1 | Replace |
| 2 | Transparent |
| 3 | XOR |
| 4 | Reverse Transparent |

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `mode` | writing mode requested |

Output Parameters:

| | |
|---|---|
| `set_mode` | writing mode selected |

`VS_COLOR ( handle index red green blue --- )`

Sets a colour index to a colour specified RGB combination.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `index` | colour index |
| `red` | red (0-1000) |
| `green` | green (0-1000) |
| `blue` | blue (0-1000) |

Output Parameters: NONE

`VSL_TYPE ( handle style --- set_type )`

Set line type for polyline operations. The styles are as follows

| | |
|---|---|
| 1 | solid |
| 2 | long dash |
| 3 | dot |
| 4 | dash,dot |
| 5 | dash |
| 6 | dash,dot,dot |
| 7 | user-defined |
| 8-n | device dependant |

Input Parameters:

| | |
|---|---|
| handle | device handle |
| style | requested line style |

Output Parameters:

| | |
|---|---|
| set_type | line style selected |

`VSL_UDSTY ( handle pattern --- )`

Set user-defined line style to 16 bit pattern word.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| pattern | line style pattern word, 16 bits |

Output Parameters: NONE

`VSL_WIDTH ( handle width --- set_width )`

Set the width of lines for poly line operations.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| width | requested line width |

Output Parameters:

| | |
|---|---|
| set_width | line width selected |

`VSL_COLOR ( handle color_index --- set_color )`

Set colour index for polyline operations.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `color_index` | requested colour index |

Output Parameters:

| | |
|---|---|
| `set_color` | line colour index |

`VSL_ENDS ( handle beg_style end_style --- )`

Sets end style of polyline.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `beg_style` | end style for beginning point<br>0 = squared (default)<br>1 = arrow<br>2 = rounded |
| `end_style` | end style for end point<br>0 = squared (default)<br>1 = arrow<br>2 = rounded |

Output Parameters: NONE

`VSM_TYPE ( handle symbol --- set_type )`

Set marker type for polymarker functions.

Marker Types:

| | | |
|---|---|---|
| 1 | . | dot |
| 2 | + | plus |
| 3 | * | asterisk |
| 4 | 0 | square |
| 5 | X | diagonal cross |
| 6 | <> | diamond |
| 7-n | | device dependant |

Input Parameters:

| | |
|---|---|
| handle | device handle |
| symbol | requested polymarker type |

Output Parameters:

| | |
|---|---|
| set_type | selected polymarker type |

```
VSM_HEIGHT ( handle height --- set_height )
```

Set polymarker height for polymarker functions.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| height | requested polymarker height |

Output Parameters:

| | |
|---|---|
| set_height | selected polymarker height |

```
VSM_COLOR ( handle color_index --- set_color )
```

Set colour index for polymarker functions

Input Parameters:

| | |
|---|---|
| handle | device handle |
| color_index | requested polymarker colour |

Output Parameters:

| | |
|---|---|
| set_color | selected polymarker height |

```
VST_HEIGHT ( handle height --- char_width
      char_height cell_width cell_height )
```

Set current graphic text character height.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| height | requested character height |

Output Parameters:

| | |
|---|---|
| char_width | character width selected |
| char_height | character height selected |
| cell_width | character cell width |
| cell_height | character cell height |

```
VST_POINT ( handle point --- set_point char_width
      char_height cell_width cell_height )
```

Set current graphic text character height in printer points. A point is 1/72 of an inch.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| height | cell height in points |

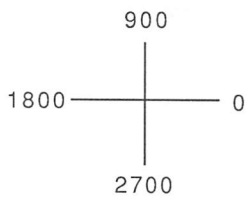
Output Parameters:

| | |
|---|---|
| set_point | character height in points |
| char_height | character height selected |
| cell_width | character cell width |
| cell_height | character cell height |

```
VST_ROTATION ( handle angle --- set_baseline )
```

Request an angle of rotation for character baseline vector

Angle spec:




Input Parameters:

| | |
|---|---|
| handle | device handle |
| angle | requested angle of rotation (0-3600) |

Output Parameters:

| | |
|---|---|
| set_baseline | selected angle of rotation (0-3600) |

`VST_FONT ( handle font --- set_font )`

Select a graphic character face for graphic text operations.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `font` | requested software text face |

Output Parameters:

| | |
|---|---|
| `set_font` | text face selected |

`VST_COLOR ( handle color_index --- set_color )`

Set colour index for graphic text operations.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `color_index` | requested text colour |

Output Parameters:

| | |
|---|---|
| `set_color` | selected text colour |

`VST_EFFECTS ( handle effect --- set_effect )`

Set text special effects for displayed graphic text.

| **Special Effect** | **Bit Map** |
|---|---|
| 0 | thickened |
| 1 | intensity |
| 2 | skewed |
| 3 | underlined |
| 4 | outline |
| 5 | shadow |

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `effect` | special effect word |

Output Parameters:

| | |
|---|---|
| `set_effect` | special effect selected |

```
VST_ALIGNMENT ( handle hor_in vert_in --- hor_out
      vert_out )
```

Set horizontal and vertical alignment for graphic text.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| hor_in | horizontal alignment requested<br>0 = left justified (default)<br>1 = centre justified<br>2 = rounded |
| vert_in | end style for end point<br>0 = squared (default)<br>1 = half line<br>2 = ascent line<br>3 = bottom<br>4 = descent<br>5 = top |

Output Parameters:

| | |
|---|---|
| hor_out | horizontal alignment selected |
| vert_out | vertical alignment selected |

```
VSF_INTERIOR ( handle style --- set_interior )
```

Set the fill interior style used in polygon operations

Input Parameters:

| | |
|---|---|
| handle | device handle |
| style | horizontal alignment requested<br>0 = hollow<br>1 = solid<br>2 = pattern<br>3 = hatch<br>4 = user defined style |

Output Parameters:

| | |
|---|---|
| set_interior | fill interior style selected |

`VSF_STYLE ( handle style_index --- set_style )`

Set the fill style based on the fill interior style.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `style_index` | requested pattern fill style index |

Output Parameters:

| | |
|---|---|
| `set_style` | pattern fill style index selected |

`VSF_COLOR ( handle color_index --- set_color )`

Set the colour index for polygon fill functions.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `color_index` | requested fill colour index |

Output Parameters:

| | |
|---|---|
| `set_style` | requested fill colour index selected |

`VSF_PERIMETER ( handle per_vis --- set_perimeter )`

Turns the outline of a fill area on and off. Default is visibility on at Open Workstation.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `style` | visibility flag<br>0 = invisible<br>+n = visible |

Output Parameters:

| | |
|---|---|
| `set_perimeter` | visibility selected |

```
VSF_UDPAT ( handle planes --- )
```

`pfill_pat` must be set up in `INTIN`.

Re-define the user definable fill pattern.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `planes` | number of planes |

Output Parameters: NONE.

## 5.4.5 GEM VDI Raster Operations

```
VRO_CPYFM ( handle wr_mode psrcMFDB pdesMFDB --- )
```

Copy a rectangular raster area from source to destination. `PTSIN` must be filled in before this call.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `wr_mode` | logic operation |
| `psrcMFDB` | address of source MFDB |
| `pdesMFDB` | address of destination MFDB |

The `PTSIN` parameters are specified as follows:

| | |
|---|---|
| 0 `PTSIN` | x-coord of corner of rectangle |
| 1 `PTSIN` | y-coord of corner of rectangle |
| 2 `PTSIN` | x-coord of diagonally opposite corner |
| 3 `PTSIN` | y-coord of diagonally opposite corner |

Output Parameters: NONE

```
VRT_CPYFM ( handle wr_mode psrcMFDB pdesMFDB
        color_index_1s color_index_0s --- )
```

Copy a monochrome rectangular raster area from source to a colour area. `PTSIN` must be filled in before this call.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `psrcMFDB` | address of source MFDB |
| `pdesMFDB` | address of destination MFDB |
| `color_index_1s` | colour index for 1s |
| `color_index_0s` | colour index for 0s |

PXYARRAY:

| | |
|---|---|
| 0 PTSIN | x-coord of corner of rectangle |
| 1 PTSIN | y-coord of corner of rectangle |
| 2 PTSIN | x-coord of diagonally opposite corner |
| 3 PTSIN | y-coord of diagonally opposite corner |

Output Parameters: NONE

```
VR_TRNFM ( handle psrcMFDB pdesMFDB --- )
```

Transform a raster area from a standard format to a device specific format or vice versa.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| psrcMFDB | address of source MFDB |
| pdesMFDB | address of destination MFDB |

Output Parameters: NONE

```
V_GET_PIXEL ( handle x y --- pel index )
```

Get a pixel value and colour index for a pixel at (x,y).

Input Parameters:

| | |
|---|---|
| handle | device handle |
| x | x-coord of pixel |
| y | y-coord of pixel |

Output Parameters:

| | |
|---|---|
| handle | device handle |
| pel | pixel value |
| index | colour index |

## 5.4.6 GEM VDI Input Functions

`VSIN_MODE ( handle dev_type mode --- )`

Set input mode for following logical input devices to request or sample.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `dev_type` | logical input device<br>1 = locator<br>2 = valuator<br>3 = choice<br>4 = string |
| `mode` | input mode<br>1 = request<br>2 = sample |

Output Parameters: NONE

`VRQ_LOCATOR ( handle x y --- xout yout term )`

Get the position of the specified locator device.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `x` | initial x-coord of locator |
| `y` | initial y-coord of locator |

Output Parameters:

| | |
|---|---|
| `x` | final x-coord of locator |
| `y` | final y-coord of locator |
| `term` | locator terminator |

`VRQ_VALUATOR ( handle valuator_in --- valuator_out terminator )`

Returns the value of the valuator device. The initial value is incremented or decremented until a terminating character is found.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `valuator_in` | initial valuator |

Output Parameters:

| | |
|---|---|
| valuator_out | valuator out |
| terminator | locator terminator |

VRQ_CHOICE ( handle ch_in --- ch_out )

Return the choice status of a selected choice device.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| ch_in | initial choice number |

Output Parameters:

| | |
|---|---|
| ch_out | choice number |

VRQ_STRING ( handle max_length echo_mode echo_x echo_y --- string )

Get a string until carriage return or INTOUT array is full.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| max_length | maximum string length |
| echo_mode | echo mode<br>0 = no echo<br>1 = echo |
| mode | input mode<br>1 = request<br>2 = sample |
| echo_x | x-coord of echo |
| echo_y | y-coord of echo |

Output Parameters:

| | |
|---|---|
| string | string in INTOUT |

`VSC_FORM ( handle pcur_form_addr --- )`

Re-define mouse cursor form. The Mouse Form is 37 words long and can be defined as

```
37 WARRAY MOUSE_FORM
```

Mouse Form

| | |
|---|---|
| 0 | x-coord hot spot |
| 1 | y-coord hot spot |
| 2 | reserved = 1 |
| 3 | mask colour index = 0 |
| 4 | data colour index = 1 |
| 5-20 | 16 words of 16 bit cursor mask |
| 21-36 | 16 words of 16 bit cursor data |

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `pcur_form_addr` | address of mouse cursor form |

Output Parameters: NONE

`VEX_TIMV ( handle tim_addr --- otim_addr tim_conv )`

Allow application to patch into timer interrupt vector and perform some action on each timer tick.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `tim_addr` | address of application timer |

Output Parameters:

| | |
|---|---|
| `otim_addr` | address of old timer |
| `tim_conv` | milliseconds per tick |

`V_SHOW_C ( handle reset --- )`

Show mouse cursor.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| reset | reset flag<br>0 = ignore number of hide calls<br>+n= normal show cursor |

Output Parameters: NONE

`V_HIDE_C ( handle --- )`

Hide mouse cursor

Input Parameters:

| | |
|---|---|
| handle | device handle |

Output Parameters: NONE

`VQ_MOUSE ( handle --- pstatus x y)`

Get the current state of the mouse buttons

Input Parameters:

| | |
|---|---|
| handle | device handle |

Output Parameters:

| | |
|---|---|
| pstatus | mouse button status |
| x | x position of cursor |
| y | y position of cursor |

`VEX_BUTV ( handle pusrcode --- psavcode )`

Allow application to patch into button change vector and perform some action each time the state of the mouse buttons change.

Input Parameters:

| | |
|---|---|
| handle | device handle |
| pusrcode | address of mouse button state change code |

Output Parameters:

| | |
|---|---|
| psavcode | address of old mouse button state change code |

`VEX_MOTV ( handle pusrcode --- psavcode )`

Allow application to patch into mouse movement vector and perform some action each time the mouse moves to a new location.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `pusrcode` | address of mouse movement state change code |

Output Parameters:

| | |
|---|---|
| `psavcode` | address of old mouse movement state change code |

`VEX_CURV ( handle pusrcode --- psavcode )`

Allow application to patch into cursor change vector and perform some action each time the cursor is drawn. The application can take over drawing of the cursor or perform some action and let GEM VDI draw cursor.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `pusrcode` | address of cursor draw code |

Output Parameters:

| | |
|---|---|
| `psavcode` | address of old cursor draw code |

`VQ_KEY_S ( handle --- pstatus )`

Get the current state of the keyboard's `control`, `shift` and `Alt` keys.

| **bit no** | **effect** |
|---|---|
| 0 | right shift key |
| 1 | left shift key |
| 2 | control key |
| 3 | Alt key |

Input Parameters:

| | |
|---|---|
| `handle` | device handle |

Output Parameters:

| | |
|---|---|
| `pstatus` | keyboard status |

## 5.4.7 GEM VDI Inquire Functions

`VQ_EXTND ( handle owflag --- )`

Returns additional device information not included in the Open Workstation call. If `owflag` = 0 then the VDI returns the same values as an `V_OPNWK` call. If `owflag` = 1 then the VDI returns extended inquire values. The values are returned in the `INTOUT` array.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `owflag` | information flag |

Output Parameters: NONE

`VQ_COLOR ( handle color_index set_flag --- rgb0 rgb1 rgb2)`

Returns either the requested or the actual value of the colour index in RGB units.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `color_index` | colour whose RGB representation is sought |
| `set_flag` | Set or actual flag<br>0=set (i.e. as requested)<br>1=actual (i.e. as shown on device) |

Output Parameters:

| | |
|---|---|
| `rgb0` | red intensity |
| `rgb1` | green intensity |
| `rgb2` | blue intensity |

`VQL_ATTRIBUTES ( handle --- )`

The current settings of all attributes affecting polylines are returned in `INTOUT` and `PTSOUT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |

Output Parameters: NONE

`VQM_ATTRIBUTES ( handle --- )`

The current settings of all attributes affecting polymarkers are returned in `INTOUT` and `PTSOUT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |

Output Parameters: NONE

`VQF_ATTRIBUTES ( handle --- )`

The current settings of all attributes affecting fill areas are returned in `INTOUT` and `PTSOUT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |

Output Parameters: NONE

`VQT_ATTRIBUTES ( handle --- )`

The current settings of all attributes affecting text items are returned in `INTOUT` and `PTSOUT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |

Output Parameters: NONE

`VQT_EXTENT ( handle string --- )`

Returns a rectangle that encloses the requested `string`. The coordinates are returned in `PTSOUT`.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `string` | address of the string |

Output Parameters: NONE

`VQIN_MODE ( handle dev_type --- input_mode)`

Returns the current input mode for the logical input device: locator, choice and string.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `dev_type` | logical input device<br>1 = locator<br>2 = valuator<br>3 = choice |

Output Parameters:

| | |
|---|---|
| `input_mode` | input mode<br>1 = request<br>2 = sample |

## 5.4.8 GEM VDI Escape Functions

`V_ESCAPES ( handle param_in verts_in func_id --- )`

The escape functions perform a wide variety of different functions depending on the device.

The input parameters must be set up in the `INTIN` and `PTSIN` arrays. Any output parameters will be returned in the `INTOUT` array.

Input Parameters:

| | |
|---|---|
| `handle` | device handle |
| `param_in` | number of parameters in `INTIN` |
| `verts_in` | number of parameters in `PTSIN` |
| `func_id` | function identifier. See table below |

Output Parameters: NONE

| | |
|---|---|
| 1 | Inquire addressable Alpha character cells |
| 2 | Exit Alpha mode |
| 3 | Enter Alpha mode |
| 4 | Alpha cursor up |
| 5 | Alpha cursor down |
| 6 | Alpha cursor right |
| 7 | Alpha cursor left |
| 8 | Home Alpha cursor |
| 9 | Erase to end of Alpha screen |
| 10 | Erase to end of Alpha text line |
| 11 | Direct Alpha cursor address |
| 12 | Output cursor addressable Alpha text |
| 14 | Reverse video off |
| 15 | Inquire current Alpha cursor address |
| 16 | Inquire tablet status |
| 17 | Hardcopy |
| 18 | Place graphic cursor at location |
| 19 | Remove last graphic cursor |
| 20 | Form advance |
| 21 | Output window |
| 22 | Clear display list |
| 23 | Output bit image file |
| 24-59 | Unused but reserved |
| 60 | Select palette |
| 61-90 | Unused but reserved |
| 91 | Inquire palette film types |
| 92 | Inquire palette driver state |
| 93 | Set palette driver state |
| 94 | Save palette driver state |
| 95 | Suppress palette inquire |
| 96 | Palette error inquire |
| 98 | Update metafile item |
| 99 | Write metafile item |
| 100 | Change GEM VDI file name |
| >100 | Unused and available for user extensions |

As an example of using the V_ESCAPES word, here are some Forth words implementing some of the escape function identifiers.

```
: VQ_CHCELLS   ( handle --- rows columns)
   0  0                    \ params = 0,   vertices = 0
   1                       \ function id
    V_ESCAPES
   0 INTOUT W@             \ rows
   1 INTOUT W@ ;           \ columns

: V_EXIT_CUR   ( handle --- )
   0   0   2   V_ESCAPES ;

: V_ENTER_CUR   ( handle --- )
   0   0   3   V_ESCAPES ;

: VS_CURADDRESS   ( handle row column --- )
   1 INTOUT W!             \ column
   0 INTOUT W!             \ row
   2  0                    \ params = 2,   vertices = 0
   11   \ function id
   V_ESCAPES ;

: VQ_CURADDRESS ( handle --- row colum)
   0  0                    \ params = 0,   vertices = 0
   15                      \ function id
    V_ESCAPES
   0 INTOUT W@             \ row
   1 INTOUT W@ ;           \ column
```

Please refer to GEM documentation in one of the books listed in the bibliography for more information on the other function identifiers.

## 5.5 GEM AES

The GEM AES is called from FORTH by placing parameters on the FORTH data stack. The parameters are in the same order as for C or Assembler. They also use the Digital Research names for the different functions.

e.g. C binding for MENU_BAR is

```
me_breturn  =  menu_bar(me_btree,  me_bshow)
```

in FORTH this would be

```
MENU_BAR  ( me_btree  me_bshow  ---  me_breturn  )
```

The above FORTH word would expect 2 values on the data stack the first the addr me_btree, the second the me_bshow value and leave one value on the stack me_breturn. Data stack before and after a MENU_BAR call

| | before | after |
|---|---|---|
| TOS | me_bshow | me_breturn |
| SOS | me_btree | |

Where it is not practical to pass all the required parameters via the data stack, it is noted under the GEM word stack picture. In this case the GEM arrays must be filled before the call with any extra values or addresses needed.

## GEM AES arrays

The following arrays are pre-defined arrays used by the GEM AES. They expect a cell number on the stack and return an address that can used like a FORTH variable. If the array is a word array then use `w!` and `w@` to store and fetch word values between the array and the stack.

If the array is a long-word array then use ! and @ to fetch long-words or 32 bit addresses between the array and the stack.

```
PB      ( n --- addr )    address array
```

This is the GEM AES Parameter array. It is pre-defined with the addresses of the other GEM arrays. `PB` Array:

| CONTROL |
|---|
| GLOBAL |
| INT_IN |
| INT_OUT |
| ADDR_IN |
| ADDR_OUT |

```
CONTROL ( n --- addr )  word array
GLOBAL  ( n --- addr )  word array
INTIN   ( n --- addr )  word array
INTOUT  ( n --- addr )  word array
ADDRIN  ( n --- addr )  long-word address array
ADDROUT ( n --- addr )  long-word address array
GEMSYS  ( --- )
```

Call GEM AES. This call needs the GEM AES arrays set up before the call.

```
>GEM    ( PBaddr --- )
```

This word is the used to pass preset AES arrays to GEM.

## 5.5.1 GEM AES Application Library Routines

`APPL_INIT ( --- ap_id )`

Initialises the application and establishes a number of GEM AES data structures.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| `ap_id` | 0 or +n = `APPL_INIT` was ok; this is placed in the GEM AES global array<br>-1 = `APPL_INIT` was not ok. |

`APPL_READ ( ap_rid ap_rlength ap_rpbuff --- ap_rreturn )`

Reads a number of bytes from a message pipe.

Input Parameters:

| | |
|---|---|
| `ap_rid` | ap_id of the message pipe to read |
| `ap_rlength` | the number of bytes to read |
| `ap_rpbuff` | address of read buffer |
| `ap_rreturn` | return message 0 = error, n= no error |

`APPL_WRITE ( ap_wid ap_wlength ap_wpbuff --- ap_wreturn )`

Writes a number of bytes to a message pipe.

Input Parameters:

| | |
|---|---|
| `ap_wid` | ap_id of the message pipe to read |
| `ap_wlength` | the number of bytes to read |
| `ap_wpbuff` | address of read buffer |

Output Parameters:

| | |
|---|---|
| `ap_wreturn` | return message 0 = error, n= no error |

`APPL_FIND ( ap_fname --- ap_fid )`

Finds the ap_id of another application in the system.

Input Parameters:

| | |
|---|---|
| `ap_fname` | name of application to find. This string must be 8 characters long. If it is shorter then it should be padded with spaces. |

Output Parameters:

| | |
|---|---|
| `ap_fid` | ap_id of found application or -1 = not found |

`APPL_TPLAY ( ap_tpmem ap_tpnum ap_tpscale --- ap_tpreturn )`

Plays a piece of a GEM AES recording of the user's actions.

Input Parameters:

| | |
|---|---|
| `ap_tpmem` | address of recording |
| `ap_tpmem` | the number of user actions to play back |
| `ap_tpscale` | a sliding scale (1 to 10,000) of play back speed<br>50 = half speed<br>100 = full speed<br>200 = twice speed |

Output Parameters:

| | |
|---|---|
| `ap_tpreturn` | always returns 1 |

```
APPL_TRECORD    ( ap_trmem ap_trcount --- ap_trreturn
      )
```

Records a users interactions with GEM AES.

Each user event uses 6 bytes as follows

| |
|---|
| • WORD code for event<br>0 = timer<br>1 = button<br>2 = mouse<br>3 = keyboard |
| • LONG value depends on event<br>timer — number of milliseconds elapsed<br>button — low WORD button up = 0, button down = 1<br>high WORD is number of clicks<br>mouse — low WORD mouse's X-coordinate in pixels<br>high WORD mouse's Y-coordinate in pixels<br>keyboard — low WORD character user typed<br>high WORD keyboard state |

Input Parameters:

| | |
|---|---|
| ap_trmem | address of recording |
| ap_trcount | the number of user actions to store |

Output Parameters:

| | |
|---|---|
| ap_trreturn | number of user events recorded |

```
APPL_EXIT    ( --- ap_xreturn )
```

Clean up after application is done.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| ap_xreturn | return message. 0 = error, n = no error |

## 5.5.2 GEM AES Event Library Routines

`EVNT_KEYBD ( --- ev_kreturn )`

Waits for a keyboard event.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| `ap_xreturn` | High WORD = scan code, low WORD = key code |

```
EVNT_BUTTON ( ev_bclicks ev_bmask ev_bstate ---
      ev_bmx ev_bmy ev_bbutton ev_bkstate ev_breturn
      )
```

Waits for a mouse button event.

Input Parameters:

| | |
|---|---|
| `ev_bclicks` | number of mouse clicks to wait for |
| `ev_bmask` | mouse buttons to wait for<br>1 = left button<br>2 = right button |
| `ev_bstate` | the button state to wait for<br>0 = down<br>1 = up |

Output Parameters:

| | |
|---|---|
| `ev_bmx` | x-coord of mouse |
| `ev_bmy` | y-coord of mouse |
| `ev_bbutton` | the mouse button event that occurred |
| `ev_bstate` | the keyboard state when event occurred.bits set as follows:<br>1 = Right shift<br>2 = Left shift<br>4 = Ctrl<br>8 = Alt |
| `ev_breturn` | number of times button entered `ev_bstate` |

```
EVNT_MOUSE ( ev_moflags ev_mox ev_moy ev_mowidth
      ev_moheight --- ev_momx ev_momy ev_mobutton
      ev_mokstate )
```

Waits for mouse event.

Input Parameters:

| | |
|---|---|
| ev_moflags | flag for call<br>0 = return on entry<br>1 = return on exit |
| ev_mox | the x-coord of mouse rectangle |
| ev_moy | the y-coord of mouse rectangle |
| ev_mowidth | the width of mouse rectangle |
| ev_moheight | the height of mouse rectangle |

Output Parameters:

| | |
|---|---|
| ev_momx | x-coord of mouse |
| ev_momy | y-coord of mouse |
| ev_mobutton | the mouse button state when the event occurred |
| ev_mokstate | the keyboard state when event occurred. Bits set as follows:<br>1 = Right shift<br>2 = Left shift<br>4 = Ctrl<br>8 = Alt |

```
EVNT_MESAG ( ev_mgpbuff --- )
```

Waits for message event.

Input Parameters:

| | |
|---|---|
| ev_mgpbuff | address of 8 word message buffer |

Output Parameters: NONE

```
EVNT_TIMER ( ev_tcount --- )
```

Waits for timer event.

Input Parameters:

| | |
|---|---|
| ev_tcount | length of time interval in milliseconds |

Output Parameters: NONE

`EVNT_MULTI ( ev_mgpbuff ev_mflags --- ev_mwhich )`

Waits for multiple events.

The `INTIN` array is set up depending on the events that are being waited for.

Input Parameters:

| | |
|---|---|
| `ev_mogpbuff` | see `EVNT_MESAG` |
| `ev_mflags` | type of events to wait for. Bit settings in hex:<br>1 keyboard<br>2 button<br>4 M1<br>8 M2<br>10 message<br>20 timer |
| `ev_bstate` | the button state to wait for<br>0 = down<br>1 = up |
| `1 INT_IN` | ev_mbclicks |
| `2 INT_IN` | ev_mbmask |
| `3 INT_IN` | ev_mbstate |
| `4 INT_IN` | ev_mm1flags |
| `5 INT_IN` | ev_mm1x |
| `6 INT_IN` | ev_mm1y |
| `7 INT_IN` | ev_mm1width |
| `8 INT_IN` | ev_mm1height |
| `9 INT_IN` | ev_mm2flags |
| `10 INT_IN` | ev_mm2x |
| `11 INT_IN` | ev_mm2y |
| `12 INT_IN` | ev_mm2width |
| `13 INT_IN` | ev_mm2height |
| `14 INT_IN` | ev_mtlocount |
| `15 INT_IN` | ev_mthicount |

Output Parameters:

| | |
|---|---|
| ev_mwhich | the events that occurred (bit settings same as for ev_mflags) |
| 1 INT_OUT | ev_mmox |
| 2 INT_OUT | ev_mmoy |
| 3 INT_OUT | ev_mmobutton |
| 4 INT_OUT | ev_mmostate |
| 5 INT_OUT | ev_mkreturn |
| 6 INT_OUT | ev_mbreturn |

```
EVNT_DCLICK ( ev_dnew ev_dgetset --- ev_dspeed )
```

Sets and gets the speed required for double-clicking.

Input Parameters:

| | |
|---|---|
| ev_dgetset | purpose of call<br>0 = get current double-click speed<br>1 = set a new double-click speed |
| ev_dnew | new double-click speed (0-4) |

Output Parameters:

| | |
|---|---|
| ev_dspeed | double-click speed |

## 5.5.3 GEM AES Menu Library Routines

```
MENU_BAR    ( me_btree me_bshow --- me_breturn )
```

Displays or erases the menu bar.

Input Parameters:

| | |
|---|---|
| me_btree | addr of menu tree |
| me_bshow | 0 = erase menu<br>1 = display menu |

Output Parameters:

| | |
|---|---|
| me_breturn | 0 = error, +n = no error |

```
MENU_ICHECK ( me_ctree  me_citem  me_ccheck  ---
      me_creturn )
```

Displays or erases a check mark next to a menu item.

Input Parameters:

| | |
|---|---|
| me_ctree | addr of menu tree |
| me_citem | number of object |
| me_ccheck | 0 = erase check mark<br>1 = display check mark |

Output Parameters:

| | |
|---|---|
| me_creturn | 0 = error, +n = no error |

```
MENU_IENABLE  ( me_etree  me_eitem  me_eenable  ---
      me_ereturn )
```

Displays an enabled item in normal brightness and a disabled item in dimmed characters.

Input Parameters:

| | |
|---|---|
| me_etree | addr of menu tree |
| me_eitem | number of object |
| me_eenable | 0 = disable item<br>1 = enable item |

Output Parameters:

| | |
|---|---|
| me_ereturn | 0 = error, +n = no error |

```
MENU_TNORMAL  ( me_ntree  me_ntitle  me_nnormal  ---
      me_nreturn )
```

Displays menu title in normal or reverse video.

Input Parameters:

| | |
|---|---|
| me_ntree | addr of menu tree |
| me_ntitle | number of title |
| me_nnormal | 0 = reverse video<br>1 = normal video |

Output Parameters:

| | |
|---|---|
| me_nreturn | 0 = error, +n = no error |

```
MENU_TEXT   ( me_ttree  me_titem  me_ttext  ---
      me_treturn )
```

Changes the text of a menu item.

Input Parameters:

| | |
|---|---|
| me_ttree | addr of menu tree |
| me_titem | number of object |
| me_ttext | address of text string for menu item |

Output Parameters:

| | |
|---|---|
| me_creturn | 0 = error, +n = no error |

```
MENU_REGISTER   ( me_rapid me_rpstring --- me_rmenuid
      )
```

Lets a desk accessory set a text string on the desk menu and obtain a desk accessory id.

Input Parameters:

| | |
|---|---|
| me_rapid | desk accessory identifier |
| me_titem | address of desk menu text string |

Output Parameters:

| | |
|---|---|
| me_rmenuid | desk menu identifier (0-5) |

## 5.5.4 GEM AES Object Library Routine

```
OBJC_ADD    ( ob_atree  ob_aparent  ob_achild  ---
      ob_areturn )
```

Add object to object tree.

Input Parameters:

| | |
|---|---|
| ob_atree | addr of object tree |
| ob_aparent | number of parent object |
| ob_achild | number of object to add |

Output Parameters:

| | |
|---|---|
| ob_areturn | 0 = error, +n = no error |

`OBJC_DELETE ( ob_dltree ob_dlobject --- ob_dlreturn )`

Delete object from object tree.

Input Parameters:

| | |
|---|---|
| `ob_dltree` | addr of object tree |
| `ob_dl_objec t` | number of parent object |
| `ob_achild` | number of object to add |

Output Parameters:

| | |
|---|---|
| `ob_dlreturn` | 0 = error, +n = no error |

`OBJC_DRAW ( ob_drtree ob_drstartob ob_drdepth --- ob_drreturn )`

Draws an object tree. Clipping of object is set in `INT_IN` array.

Input Parameters:

| | |
|---|---|
| `ob_drtree` | addr of object tree |
| `ob_drstartob` | object to start drawing from |
| `ob_drdepth` | level of depth to draw to |

Output Parameters:

| | |
|---|---|
| `ob_drreturn` | 0 = error, +n = no error |

`OBJC_FIND ( ob_ftree ob_fstartob ob_fdepth fmx fmy --- ob_freturn )`

Find object under mouse.

Input Parameters:

| | |
|---|---|
| `ob_ftree` | addr of object tree |
| `ob_fstartob` | object to start searching from |
| `ob_fdepth` | number of levels to search |
| `ob_fmx` | x-coord of mouse |
| `ob_fmy` | y-coord of mouse |

Output Parameters:

| | |
|---|---|
| `ob_fobnum` | number of object under mouse (-1 = no object) |

```
OBJC_OFFSET ( ob_oftree ob_ofobject --- ob_ofreturn
      ob_ofxoff ob_ofyoff )
```

Calculates the coordinates of object relative to screen origin.

Input Parameters:

| | |
|---|---|
| ob_oftree | addr of object tree |
| ob_ofobject | object number |

Output Parameters:

| | |
|---|---|
| ob_fobnum | number of object under mouse (-1 = no object) |
| ob_ofxoff | x-coord of object |
| ob_ofyoff | y-coord of object |

```
OBJC_ORDER ( ob_ortree ob_orobject ob_ornewpos ---
      ob_orreturn )
```

Re-order object in object tree.

Input Parameters:

| | |
|---|---|
| ob_ortree | addr of object tree |
| ob_orobject | object to order |
| ob_ornewpos | new position of object |

Output Parameters:

| | |
|---|---|
| ob_orreturn | 0 = error +n=no_error |

```
OBJC_EDIT ( ob_edtree ob_edobject ob_edchar
      ob_edidx ob_edkind --- ob_return ob_ednewidx )
```

Edit text in G_TEXT or G_BOXTEXT type object.

Input Parameters:

| | |
|---|---|
| ob_edtree | address of object tree |
| ob_edobject | object to edit |
| ob_edchar | character input of user |
| ob_edidx | next character position |
| ob_edkind | editor functions (1-3) |

Output Parameters:

| | |
|---|---|
| ob_edreturn | 0 = error, +n = no error |
| ob_ednewiddx | next character position after OBJC_EDIT |

```
OBJC_CHANGE ( ob_ctree ob_cobject ob_cxclip ob_cyclip
      ob_cwclip ob_chclip ob_cnewstate ob_credraw --
      - ob_creturn )
```

Input Parameters:

| | |
|---|---|
| ob_ctree | address of object tree |
| ob_cobject | object to change |
| ob_cxclip | x-coord of clip rectangle |
| ob_cyclip | y-coord of clip rectangle |
| ob_cwclip | width of clip rectangle |
| ob_chclip | height of clip rectangle |
| ob_cnewstate | new status of object |
| ob_credraw | 0 = no redraw, 1 = redraw |

Output Parameters:

| | |
|---|---|
| ob_creturn | 0 = error, +n = no error |

## 5.5.5 GEM AES Form Library Routines

```
FORM_DO ( fo_dotree fo_dostartob --- fo_doreturn )
```

Causes the form library to monitor the user's interaction with a form.

Input Parameters:

| | |
|---|---|
| fo_dotree | address of object tree |
| fo_dostartob | first text field to edit, 0 for no start object |

Output Parameters:

| | |
|---|---|
| fo_doreturn | 0 = error, +n = no error |

`FORM_DIAL ( fo_diflag --- fo_direturn )`

Reserves or frees the portion of the screen used for dialog boxes and can draws an expanding or shrinking box. The INT_IN array holds small and large rectangles for FMD_GROW and FMD_SHRINK.

Input Parameters:

| | |
|---|---|
| `fo_diflag` | FORM_DIAL action<br>0 `FMD_START` reserve screen space<br>1 `FMD_GROW` draws expanding box<br>2 `FMD_SHRINK` draws shrinking box<br>3 `FMD_FINISH` frees screen space and cause redraw |
| `fo_dostartob` | first text field to edit, 0 for no start object |
| 1 `INT_IN` | fo_dilittlx |
| 2 `INT_IN` | fo_dilittly |
| 3 `INT_IN` | fo_dilittlw |
| 4 `INT_IN` | fo_dilittlh |
| 5 `INT_IN` | fo_dibigx |
| 6 `INT_IN` | fo_dibigy |
| 7 `INT_IN` | fo_dibigw |
| 8 `INT_IN` | fo_dibigh |

Output Parameters:

| | |
|---|---|
| `fo_doreturn` | 0 = error, +n = no error |

`FORM_ALERT ( fo_adefbttn fo_astring --- fo_aexbttn )`

Displays an alert box.

Input Parameters:

| | |
|---|---|
| `fo_adefbttn` | forms default exit button<br>0 no default<br>1 first exit button<br>2 second exit button<br>3 third exit button |
| `fo_astring` | address of string containing alert |

Output Parameters:

| | |
|---|---|
| `fo_aexbttn` | number indicating exit button selected |

```
FORM_ERROR  ( fo_enum --- fo_eexbttn )
```

Displays an error box.

Input Parameters:

| | |
|---|---|
| fo_enum | DOS Error code |

Output Parameters:

| | |
|---|---|
| fo_eexbttn | number indicating exit button selected |

```
FORM_CENTER ( fo_ctree --- )
```

Centres a dialog box on the screen.

Input Parameters:

| | |
|---|---|
| fo_ctree | address of object tree dialog |

Output Parameters: NONE.

The INT_OUT array contains

| | |
|---|---|
| 1 INT_OUT | x-coord of centred object |
| 2 INT_OUT | y-coord of centred object |
| 3 INT_OUT | width of centred object |
| 4 INT_OUT | height of centred object |

## 5.5.6 GEM AES Graphics Library

```
GRAF_RUBBERBOX (   gr_rx    gr_ry    gr_rminwidth
      gr_rminheight  ---  gr_rreturn  gr_rlastwidth
      gr_rlastheight )
```

Draws a rubber box that expands and contracts from a fixed point as the mouse moves.

Input Parameters:

| | |
|---|---|
| gr_rx | x-coordinate of box |
| gr_ry | y-coordinate of box |
| gr_rminwidth | smallest width |
| gr_rminheight | smallest height |

Output Parameters:

| | |
|---|---|
| gr_rreturn | 0 = error, +n = no error |
| gr_rlastwidth | width of box |

| | |
|---|---|
| gr_rlastheight | height of box |

```
GRAF_DRAGBOX    (  gr_dwidth  gr_dheight  gr_dstartx
      gr_dstarty  gr_dboundx  gr_dboundy  gr_dboundw
      gr_dboundh    ---    gr_dreturn    gr_dfinishx
      gr_dfinishy )
```

Moves a box, keeping the mouse pointer in the same position in the box.

Input Parameters:

| | |
|---|---|
| gr_dwidth | width of drag box |
| gr_dheight | height of drag box |
| gr_dstartx | start x-coord |
| gr_dstarty | start y-coord |
| gr_dboundx | x-coord of boundary rectangle |
| gr_dboundy | y-coord of boundary rectangle |
| gr_dboundw | width of boundary rectangle |
| gr_dboundh | height of boundary rectangle |

Output Parameters:

| | |
|---|---|
| gr_dreturn | 0 = error, +n = no error |
| gr_dfinishx | x co-ord of box |
| gr_dfinishy | y co-ord of box |

```
GRAF_MOVEBOX    (  gr_mwidth  gr_mheight  gr_msourcex
      gr_msourcey gr_mdestx gr_mdesty --- gr_mreturn
      )
```

Draws a moving box.

Input Parameters:

| | |
|---|---|
| gr_mwidth | width of box |
| gr_mheight | height of box |
| gr_msourcex | initial x-coord of box |
| gr_msourcey | initial y-coord of box |
| gr_mdestx | final x-coord of box |
| gr_mdesty | final y-coord of box |

Output Parameters:

| | |
|---|---|
| gr_mreturn | 0 = error, +n = no error |

```
GRAF_GROWBOX   (   gr_gstx   gr_gsty   gr_gstwidth
      gr_gstheight  gr_gfinx  gr_gfiny  gr_gfinwidth
      gr_gfinheight  ---  gr_greturn )
```

Draws an expanding box outline.

Input Parameters:

| | |
|---|---|
| gr_gstx | initial x-coord |
| gr_gsty | initial y-coord |
| gr_gstwidth | initial width |
| gr_gstheight | initial height |
| gr_gfinx | final x-coord |
| gr_gfiny | final y-coord |
| gr_gfinwidth | final width |
| gr_gfinheight | final height |

Output Parameters:

| | |
|---|---|
| gr_greturn | 0 = error, +n = no error |

```
GRAF_SHRINKBOX  (  gr_sfinx  gr_sfiny  gr_sfinwidth
      gr_sfinheight  gr_sstx  gr_ssty  gr_sstwidth
      gr_sstheight  ---  gr_sreturn )
```

Draws a shrinking box outline.

Input Parameters:

| | |
|---|---|
| gr_sstx | initial x-coord |
| gr_ssty | initial y-coord |
| gr_sstwidth | initial width |
| gr_sstheight | initial height |
| gr_sfinx | final x-coord |
| gr_sfiny | final y-coord |
| gr_sfinwidth | final width |
| gr_sfinheight | final height |

Output Parameters:

| | |
|---|---|
| gr_sreturn | 0 = error, +n = no error |

`GRAF_WATCHBOX ( gr_wptree gr_wobject gr_winstate gr_woutstate --- gr_wreturn )`

Watches a box to see if the mouse pointer is inside.

Input Parameters:

| | |
|---|---|
| `gr_wptree` | address of object tree containing box |
| `gr_wobject` | index of object in tree |
| `gr_winstate` | box state when mouse pointer (with button down) is inside it:<br>$00 NORMAL<br>$01 SELECTED<br>$02 CROSSED<br>$04 CHECKED<br>$08 DISABLED<br>$10 OUTLINED<br>$20 SHADOWED |
| `gr_woutstate` | box state when mouse pointer ( button down) is outside it. Values as for `gr_winstate`. |

Output Parameters:

| | |
|---|---|
| `gr_wreturn` | mouse pointer position<br>0 - outside box<br>1 - inside box |

`GRAF_SLIDEBOX ( gr_slptree gr_slparent gr_slobject gr_slvh --- gr_slreturn )`

Keeps a sliding box inside its parent box.

Input Parameters:

| | |
|---|---|
| `gr_slptree` | tree containing objects |
| `gr_slparent` | index of parent in tree |
| `gr_slobject` | index of slider in tree |
| `gr_slvh` | direction of slider movement<br>0 - horizontal<br>1 - vertical |

Output Parameters:

| | |
|---|---|
| `gr_slreturn` | position of center of slider relative to its parent (0 to 1000)<br>0 = left or top, 1000 = right or bottom |

`GRAF_HANDLE ( --- gr_handle )`

Returns a GEM VDI handle for the opened screen workstation that the GEM AES libraries use.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| `gr_handle` | GEM VDI handle |

Information about the system font is also returned by `GRAF_HANDLE` in the `INT_OUT` array, as below:

| | |
|---|---|
| 1 `INT_OUT` | gr_hwchar |
| 2 `INT_OUT` | gr_hhchar |
| 3 `INT_OUT` | gr_hwbox |
| 4 `INT_OUT` | gr_hhbox |

`GRAF_MOUSE ( gr_mofaddr gr_monumber --- gr_moreturn )`

Lets an application change the mouse form to one of a predefined set or a application defined form.

Input Parameters:

| | |
|---|---|
| `gr_mofaddr` | address of 35 word buffer that contains the mouse definition block |
| `gr_monumber` | 0 - arrow<br>1 - text cursor<br>2 - hourglass<br>3 - hand with pointing finger<br>4 - flat hand, extended fingers<br>5 - thin cross hair<br>6 - thick cross hair<br>7 - outline cross hair<br>255 - mouse form stored in `gr_mofaddr`<br>256 - hide mouse form<br>257 - show mouse form |

Output Parameters:

| | |
|---|---|
| `gr_moreturn` | 0 = error, +n = no error |

```
GRAF_MKSTATE    ( ---  gr_mkmx  gr_mkmy  gr_mkmstate
      gr_mkkstate )
```

Returns the current mouse location, mouse button state and keyboard state.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| gr_mkmx | current mouse x-coord |
| gr_mkmy | current mouse y-coord |
| gr_mkmstate | current mouse button state<br>1 - left button<br>2 - right button<br>3 - right and left buttons |
| gr_mkkstate | current keyboard state<br>bit set: 0 = key up, 1 = key down<br>1 - right-shift<br>2 - left-shift<br>4 - Ctrl<br>8 - Alt |

## 5.5.7 GEM AES Scrap Library Routines

```
SCRP_READ   ( sc_rpscrap --- sc_rreturn )
```

Reads the current scrap directory for the clipboard.

Input Parameters:

| | |
|---|---|
| sc_rpscrap | address of buffer for scrap directory |

Output Parameters:

| | |
|---|---|
| sc_rreturn | 0 = error, +n = no error |

```
SCRP_WRITE  ( sc_wpscrap --- sc_wreturn )
```

Changes the current scrap directory for the clipboard.

Input Parameters:

| | |
|---|---|
| sc_wpscrap | address of new scrap directory |

Output Parameters:

| | |
|---|---|
| sc_wreturn | 0 = error, +n = no error |

## 5.5.8 GEM AES File Selector Library

`FSEL_INPUT ( fs_iinpath fs_iinsel --- fs_ireturn )`

Displays the File Selector dialog box and lets the user select a filename.

Input Parameters:

| | |
|---|---|
| `fs_iinpath` | address of directory path |
| `fs_iinsel` | address of file name |

Output Parameters:

| | |
|---|---|
| `fs_ireturn` | 0 = error, +n = no error |

## 5.5.9 GEM AES Window Library Routines

`WIND_CREATE ( wi_crkind wi_crwx wi_crwy wi_crww wi_crwh --- wi_crreturn )`

Allocates the application's full-size window and returns a handle.

Input Parameters:

| | |
|---|---|
| `wi_crkind` | window components. Component bits:<br>1 NAME<br>2 CLOSE<br>4 FULL<br>5 MOVE<br>10 INFO<br>20 SIZE<br>40 UPARROW<br>80 DNARROW<br>100 VSLIDE<br>200 LFARROW<br>400 RTARROW<br>800 HSLIDE |
| `wi_crwx` | x-coord of full-size window |
| `wi_crwy` | y-coord of full-size window |
| `wi_crww` | width of full-size window |
| `wi_crwh` | height of full-size window |

Output Parameters:

| | |
|---|---|
| `wi_creturn` | the window handle (0-n)<br>-n = no more windows |

```
WIND_OPEN   ( wi_ohandle wi_owx wi_owy wi_oww wi_owh
    --- wi_oreturn )
```

Opens the created window to a specified size.

Input Parameters:

| | |
|---|---|
| wi_ohandle | window handle |
| wi_owx | x-coord of initial window size |
| wi_owy | y-coord of initial window size |
| wi_oww | width of initial window size |
| wi_owh | height of initial window size |

Output Parameters:

| | |
|---|---|
| wi_oreturn | 0 = error, +n = no error |

```
WIND_CLOSE  ( wi_clhandle --- wi_clreturn )
```

Close an open window.

Input Parameters:

| | |
|---|---|
| wi_clhandle | window handle |

Output Parameters:

| | |
|---|---|
| wi_clreturn | 0 = error, +n = no error |

```
WIND_DELETE ( wi_dhandle --- wi_dreturn )
```

De-allocates the application's window and handle.

Input Parameters:

| | |
|---|---|
| wi_dhandle | window handle |

Output Parameters:

| | |
|---|---|
| wi_dreturn | 0 = error, +n = no error |

`WIND_GET ( wi_ghandle wi_gfield --- wi_greturn )`

Gets information on a particular window. The results are returned in the `INT_OUT` array.

Input Parameters:

| | |
|---|---|
| `wi_ghanslw` | window handle |
| `wi_gfield` | window field<br>4 WF_WORKXYWH x,y,width,height<br>5 WF_CURRXYWH x,y,width,height<br>6 WF_PREVXYWH x,y,width,height<br>7 WF_FULLXYWH x,y,width,height<br>8 WF_HSLIDE gw1 = position (0-1000)<br>9 WF_VSLIDE gw1 = position (0-1000)<br>10 WF_TOP gw1 = active window<br>11 WF_FIRSTXYWH x,y,width,height<br>12 WF_NEXTXYWH x,y,width,height<br>15 WF_HLSIZE gw1 = size (1-1000)<br>-1 (default minimum)<br>16 WF_VLSIZE gw1 = size (1-1000)<br>-1 (default minimum) |

Output Parameters:

| | |
|---|---|
| `wi_greturn` | 0 = error, +n = no errors |

Output Fields:

| | |
|---|---|
| 2 `INT_OUT` | wi_gw1 |
| 3 `INT_OUT` | wi_gw2 |
| 4 `INT_OUT` | wi_gw3 |
| 5 `INT_OUT` | wi_gw4 |

`WIND_SET ( wi_shandle wi_sfield --- wi_sreturn )`

Sets new values for the fields that determine how a window is displayed.

Input Parameters:

| | |
|---|---|
| `wi_shandle` | window handle |
| `wi_sfield` | window field<br>1 WF_KIND `sw1` see WIND_CREATE<br>2 WF_NAME address in `sw1` and `sw2`<br>3 WF_INFO address in `sw1` and `sw2`<br>4 WF_WORKXYWH x,y,width,height<br>5 WF_CURRXYWH x,y,width,height<br>8 WF_HSLIDE sw1 = position (0-1000)<br>9 WF_VSLIDE sw1 = position (0-1000)<br>10 WF_TOP sw1 = active window<br>14 WF_NEWDESK new GEM Desktop<br>`sw1` and `sw2` = address<br>`sw3` = starting object<br>15 WF_HLSIZE `sw1` = size (1-1000)<br>-1 (default minimum)<br>16 WF_VLSIZE `sw1` = size (1-1000)<br>-1 (default minimum) |

Input Fields:

| | |
|---|---|
| 2 INT_IN | wi_sw1 |
| 3 INT_IN | wi_sw2 |
| 4 INT_IN | wi_sw3 |
| 5 INT_IN | wi_sw4 |

Output Parameters:

| | |
|---|---|
| `wi_greturn` | 0 = error, +n = no errors |

`WIND_FIND ( wi_fmx wi_fmy --- wi_freturn )`

Finds which window is under the mouse's X,Y postion.

Input Parameters:

| | |
|---|---|
| `wi_fmx` | x-coord of mouse |
| `wi_fmy` | y-coord of mouse |

Output Parameters:

| | |
|---|---|
| `wi_freturn` | 0 = error, +n = no error |

`WIND_UPDATE ( wi_ubegend --- wi_ureturn )`

Tell GEM AES that the application is about to update or has finished updating a window or that the application is about to take control of the mouse.

Input Parameters:

| | |
|---|---|
| `wi_ubegend` | call action<br>0 - END_UPDATE<br>1 - BEG_UPDATE<br>2 - END_MCTRL<br>3 - BEG_MCTRL |
| `wi_ureturn` | y-coord of mouse |

Output Parameters:

| | |
|---|---|
| `wi_ureturn` | 0 = error, +n = no error |

`WIND_CALC ( wi_ctype wi_ckind wi_cinx wi_ciny wi_cinw wi_cinh --- wi_coutx wi_couty wi_coutw wi_couth wi_creturn )`

Calculates the X and Y coordinates and the width and height of a window's work area or border.

Input Parameters:

| | |
|---|---|
| `wi_ctype` | type of calculation<br>0 = border area<br>1 = work area |
| `wi_ckind` | window components see `WIND_CREATE` |
| `wi_cinx` | input x-coord |
| `wi_ciny` | input y-coord |
| `wi_cinw` | input width |
| `wi_cinh` | input height |

Output Parameters:

| | |
|---|---|
| `wi_coutx` | output x-coord |
| `wi_couty` | output y-coord |
| `wi_coutw` | output width |
| `wi_couth` | output height |
| `wi_creturn` | 0 = error, +n = no error |

## 5.5.10 GEM AES Resource Library Routines

`RSRC_LOAD ( re_lpfname --- re_lreturn )`

Loads an entire resource file into memory.

Input Parameters:

| | |
|---|---|
| `re_lpfname` | address of resource file name |

Output Parameters:

| | |
|---|---|
| `re_lreturn` | 0 = error, +n = no error |

`RSRC_FREE ( --- re_freturn )`

Frees memory allocated during `RSRC_LOAD`.

Input Parameters: NONE

Output Parameters:

| | |
|---|---|
| `re_lreturn` | 0 = error, +n = no error |

`RSRC_GADDR ( re_gtype re_gindex --- re_greturn re_gaddr )`

Gets the address of a data structure in memory.

Input Parameters:

| | |
|---|---|
| `re_gtype` | type of data structure<br>0 = tree<br>1 = OBJECT<br>2 = TEDINFO<br>3 = ICONBLK<br>4 = BITBLK<br>5 = string<br>6 = image data<br>7 = obspec |
| `re_gindex` | index of data structure |

Output Parameters:

| | |
|---|---|
| `re_gaddr` | address of data structure |
| `wi_ureturn` | 0 = error, +n = no error |

```
RSRC_SADDR ( re_stype re_sindex --- re_sreturn
      re_saddr )
```

Stores an index to a data structure.

Input Parameters:

| | |
|---|---|
| re_stype | type of data structure<br>0 = tree<br>1 = OBJECT<br>2 = TEDINFO<br>3 = ICONBLK<br>4 = BITBLK<br>5 = string<br>6 = image data<br>7 = obspec |
| re_sindex | index of data structure |

Output Parameters:

| | |
|---|---|
| re_saddr | address of data structure |
| re_sreturn | 0 = error, +n = no error |

```
RSRC_OBFIX ( re_otree re_oobject --- )
```

Converts an object's X and Y coordinates, width and height from character coordinates to pixel coordinates.

Input Parameters:

| | |
|---|---|
| re_otree | address of object tree |
| re_oobject | index of object to be converted |

Output Parameters: NONE

## 5.5.11 GEM AES Shell Library Routines

```
SHEL_READ ( sh_rpcmd sh_rptail --- sh_rreturn )
```

Lets an application determine how it was invoked.

Input Parameters:

| | |
|---|---|
| sh_rptail | address of command tail |
| sh_rpcmd | address of command |

Output Parameters:

| | |
|---|---|
| sh_rreturn | 0 = error, +n = no error |

```
SHEL_WRITE ( sh_wdoex sh_wisgr sh_wiscr sh_wpcmd
       sh_wptail sh_wreturn )
```

Exits GEM AES or tells which application to run next.

Input Parameters:

| | |
|---|---|
| sh_wdoex | 0 = return to desktop<br>1 = run application |
| sh_wisgr | 0 = text app, 1 = graphic app |
| sh_wiscr | 0 = TOS app, 1 = GEM app |
| sh_wpcmd | address of command file |
| sh_wptail | address of command tail |

Output Parameters:

| | |
|---|---|
| sh_rreturn | 0 = error, +n = no error |

```
SHEL_FIND  ( sh_fpbuff --- sh_freturn )
```

Locates a filename by following the AES search path.

Input Parameters:

| | |
|---|---|
| sh_fpbuff | file name buffer |

Output Parameters:

| | |
|---|---|
| sh_freturn | 0 = error, +n = no error |

```
SHEL_ENVRN ( sh_epvalue sh_eparm --- )
```

Searches the DOS environment for a parameter and returns the address of its value.

Input Parameters:

| | |
|---|---|
| sh_epvalue | pointer to address of byte following parameter |
| sh_eparm | parameter string address |

Output Parameters: NONE

# Appendix A. Implementation details

## A.1 Memory Map

| | | |
|---|---|---|
| High TPA :<br>free ram used by GEMDOS Operating system | | HIMEM |
| ram disk (optional) | | LOMEM |
| Free User RAM | | HERE |
| Disk Buffers 1028 * 2 = | 2056 bytes | |
| Return Stack | 2k bytes | |
| Data Stack | 256 bytes | |
| Terminal buffer | 1360 bytes | |
| User Page | 512 bytes | |
| **HiSoft FORTH**<br>Main Kernel | | |
| Low TPA:<br>desk accessories, ram disks etc. | | |

## A.2 HiSoft FORTH Compiler

There are a number of different ways to implement the FORTH language. The traditional method has been to use a little interpreter to read the address of the next word to execute. This has the advantage of producing compact code, but the code runs slower than a truly compiled language because of the interpreter overhead.

**HiSoft FORTH** uses the *Subroutine Threaded* method, which produces a sub-routine call to the machine code for each word giving a larger but much faster program. Fortunately this increase in code size of older versions of FORTH isn't much of a problem as the ST has much more memory than, say Z80 or 6502 computers.

The header structure for **HiSoft FORTH** is different because of the subroutine threading as the CFA is not a pointer but the actual start of FORTH code and the PFA is only valid with variables and data structures.

When FORTH is compiling ( usually between : and ; ), it compiles a JSR.L CFA_of_FORTH_WORD (jump-to-subroutine instruction) for each FORTH word until it reaches a ; when it compiles an RTS opcode. So that when the compiled word is executed it makes a series of jumps to different FORTH words and executes them. When it reaches an RTS opcode it execution ends and the program continues from the next instruction.

A FORTH compiled word is called a Secondary. The words in the FORTH kernel (main body of words) could be Secondaries or Primitives. A Primitive is a FORTH code word that is written in machine code and is executed directly. A secondary could call another secondary or a primitive, but calls to secondaries words eventually call a primative to execute some machine code.

# A.3 HiSoft FORTH Headers

Each FORTH word has a header usually made by the FORTH word `CREATE`. This header comprises of a the name and length of the name found at the NFA (name field address), a link back to the previous word found at the LFA (link field address), a pointer to a code field found at the CFA (code field address), and a place where code or data starts found at the PFA (parameter field address).

e.g. Header compiled by `CREATE <NAME>`

| | |
|---|---|
| 4 bytes | Locate field (only compiled code) |
| 4 bytes | LFA link field address (relative) |
| 1 byte | SYS used by system |
| 1 byte | NFA length byte |
| "N" | NFA. The FORTH word `NAME` is here.This is padded out to be an even length. |
| "A" | |
| "M" | |
| "E" | |

CFA code starts here.

Execute needs the CFA to jump to FORTH code. Each FORTH word ends with an `RTS` 68000 opcode (Return from Subroutine).

```
42 VARIABLE FRED

' FRED PFA @ . should print 42
```

# Appendix B FIG (FORTH Interest Group)

The FORTH Interest Group is a useful source of FORTH programs and articles on FORTH. Membership is £10 per year at the time of writing and applications for membership, which includes a monthly magazine and access to an excellent reference library, should be sent to:

FIG (UK)
Membership Secretary
88 Woosehill Lane
Wokingham
BERKSHIRE RG11 2TS

The group also meets at 7pm on the first Thursday of each month at Polytechnic of South Bank Rm. 408, Borough Rd.

The address of FIG in the USA is:

FORTH Interest Group
P.O. Box 8231
San Jose, CA 95155

Membership in Europe is $42 per year.

# Appendix C
# Technical Support

## C.1 Technical Support

So that we can maintain the quality of our technical support service we are detailing how to take best advantage of it. These guidelines will make it easier for us to help you, fix bugs as they get reported and save other users from having the same problem. Technical support is available in five ways:

**CIX™** our username is (not surprisingly) *hisoft*. We also have our own conference just *j hisoft*. This is the best method as the author of the FORTH, Henry McGeough is available as *hmcg*. The Forth Interest Group also have a conference.

**Phone** We can offer limited technical support for this product during our technical hour between 3pm and 4pm, though non-European customers' calls will be accepted at other times.

**Post** if sending a disk, *please* put your name & address on it.

**BIX™** our username is (still not surprisingly) *hisoft*. Would UK customers please use CIX ; it's cheaper for everyone.

**GEnie™** our username is (yet again) *hisoft*.

For bug reports, *please* always quote the version number of the program (as given when **HiSoft FORTH** loads) and the serial number found on your master disk.

If you think you have found a bug, try and test it with a simple case. It is always easier for us to answer your questions if you send us a letter and, if the problem is with a particular program, enclose a copy on disk (which we will return).

## C.2 Upgrades

As with all our products, **HiSoft FORTH** is undergoing continual development and, periodically, new versions become available. We make a small charge for upgrades, though if extensive additional documentation is supplied the charge may be higher. All users who return their registration cards will be notified of *major* upgrades.

# Bibliography

## FORTH Books

This manual is not intended as a full tutorial. Whilst the reference sections contain all the information that is included with FORTH implementations, users who are not experienced FORTH programmers are strongly recommended to purchase one or more of the introductory texts listed below.

| Title | Author | Publisher |
|---|---|---|
| Starting FORTH | Leo Brodie | Prentice-Hall |
| Thinking FORTH | Leo Brodie | Prentice-Hall |
| Discover FORTH | Tom Hogan | McGraw-Hill |
| Introduction to FORTH | K Knecht | H W Sams |
| FORTH Programming | L J Scanlon | H W Sams |
| System Guide to FORTH | C H Ting | Offette Enterprise |
| The Complete FORTH | A Winfield | Sigma |
| FORTH Techniques | R Olney & M Benson | Pan Books |
| Dr.Dobb's Toolbook of FORTH | M Ouverson | M &T books |
| Object-Oriented Forth Implementation of Data Structures | Dick Pountain | Academic Press |
| Object-Oriented Forth | Dick Pountain | BYTE August 1986 |

These books may be obtained from many good technical bookshops, including Foyles, Blackwells and Heffers.

## ST Books

Most of these book deal with programming in C but the names of the FORTH routines (and even the parameters) are the same as the C ones.

| Title | Publisher |
|---|---|
| Compute!'s ST Applications Guide: Programming in C | Compute! Books |
| Compute!'s Guide to the Atari ST Vols 1,2,3 | Compute! Books |
| Atari ST Internals | Abacus |
| GEM on the Atari ST | Abacus |
| Tricks and Tips on the Atari ST | Abacus |
| Programmer's Guide to GEM | Sybex |

# Index


!CSP 49
!RSP 57
$CD 34
$FLOAD 36
$INCLUDE 36
$MAPS 35
$OPEN 35
* 42
*/ 42, 65
+ 42
+! 42
+C! 42
+LOAD 42
+LOOP 42, 67
+THRU 42
+W! 42
, 41
- 43
-DUP 51
-TRAILING 43
. 43
." 43
." 64
.( 43, 64
.NAME 67
.R 43
.S 43
/ 43, 65
/MOD 43, 65
0< 45
0= 45
0> 45
1+ 46
1- 46
2* 46
2+ 46
2- 46
2/ 46
2! 70
2@ 70
2CONSTANT 70
2DROP 70
2DUP 70
2OVER 70
2ROT 70
2SWAP 70
2VARIABLE 70
79-STANDARD 61
; 42,64
< 44, 45
<< 44
<> 45
<BUILDS 44
<MARK 44
<RESOLVE 44
= 45
> 45
>< 44
>> 44
>BODY 44
>GEM 125
>IN 61
>LINK 45
>M 79
>MARK 44
>NAME 45
>R 45
>RESOLVE 44
>VDI 94
>WR 45
?CSP 49
?DUP 51, 61
?MI 79
?MIDI 79
?MO 79
?RAM 33
@ 45
@RSP 57
[ 60
['] 60, 65
] 60
A-LINE 80
ABORT 46,65
ABS 46
ACTIVATE 78
ADDRIN 125
ADDROUT 125
AGAIN 46
ALLOT 46
ALSO 46
AND 46
APPL_EXIT 128
APPL_FIND 127
APPL_INIT 126
APPL_READ 126
APPL_TPLAY 127
APPL_TRECORD 128
APPL_WRITE 126
ARRAY 47
ASCII 47
ASK 47
ASM.SEQ 4
ASSEMBLER 47
AT 47
AUX 47
back-up **2**
BASE 47
BCONIN 88

BCONOUT 88
BCONSTAT 88
BEGIN 47
BIOSKEYS 91
BK 80
BKGND 82
BLANK 61
BLANKS 48
BLK 48
BLK.SEQ 4
BLK>SEQ 37
BLOCK 48
BODY> 48
BOOT 69
BOX.SEQ 4
BUFFER 48
BX 68
C 41
C! 41
C@ 45
CARRAY 47
CASE 48
CAUXIN 83
CAUXIS 84
CAUXOS 84
CAUXOUT 84
CCONIN 83
CCONIS 84
CCONOS 84
CCONOUT 83
CCONRS 84
CCONWS 84
CFA 48, 154
CHAIN 68
CLIP 81
CLREOL 48
CLREOP 48
CMOVE 49
CMOVE> 65
CNECIN 84
code field address 154
COLOR 81
COMPILE 49
COMPILE] 61
CON 49
CONSTANT 49
CONTROL 94, 125
CONVERT 49, 61
COS 81
COUNT 49
CPRNOS 84
CPRNOUT 84
CR 49
CRAWCIN 84
CRAWIO 84
CREATE 49, 61
CSP 49
CSROFF 81
CSRON 81
CURSCONF 91
D+ 71
D+- 71
D- 71
D. 71
D.R 71
D0= 71
D2/ 71
D< 71
D= 71
DABS 71
DATE 25
DCHDIR 85
DECIMAL 49
DEFER 50
DEFINITIONS 50
DEPTH 50
desk accessory 31, 36
DESKTOP 68
DevpacST 76
DFREE 85
DGETDIR 85
DGETDRV 85
dictionary 7
DISK 34
Disk Contents **4**
Disk Map 34
Disk System 33
DMAX 71
DMIN 71
DMKDIR 85
DNEGATE 71
DO 50, 66
DOES> 50
DOSOUND 92
DRAW 80
DRMDIR 85
DSETDRV 85
ED 25
EDITOR 51
ELSE 51
EMIT 51
EMPTY-BUFFERS 51
ENDCASE 51
ENDOF 51
ERASE 51
Escape Functions 122
EVNT_BUTTON 129
EVNT_DCLICK 132
EVNT_KEYBD 129
EVNT_MESAG 130
EVNT_MOUSE 130
EVNT_MULTI 131
EVNT_TIMER 130
EXECUTE 51

EXPECT 51
FATTRIB 87
FCLOSE 86
FCREATE 86
FD 80
FDATIME 87
FDELETE 86
FEDIT 36
FFORCE 87
FGETDTA 86
FGND 82
FILE 34
FILL 52
FIND 52, 62, 66
FLOAD 36
FLOAT.SEQ 4
FLOPFMT 90
FLOPRD 90
FLOPVER 91
FLOPWR 90
FLUSH 52
FNAME 52
FOPEN 86
FORGET 52, 69
FORM_ALERT 138
FORM_CENTER 139
FORM_DIAL 138
FORM_DO 137
FORM_ERROR 139
FORTH 52
FORTH Headers 154
FORTH-79 61
FORTH-83 52, 63
FORTH.BLK 4
FORTH.PRG 4
FREAD 86
FRENAME 87
FSEEK 87
FSEL_INPUT 145
FSETDTA 85
FSFIRST 87
FSNEXT 87
FWRITE 86
GEM 52
GEM VDI **93**
GEM VDI arrays 93
GEMSYS 125
GETBPB 89
GETMBP 88
GETREZ 90
GETTIME 91
GIACCESS 92
GLOBAL 125
Glossary 41
GRAF_DRAGBOX 140
GRAF_GROWBOX 141
GRAF_HANDLE 143
GRAF_MKSTATE 144
GRAF_MOUSE 143
GRAF_MOVEBOX 140
GRAF_RUBBERBOX 139
GRAF_SHRINKBOX 141
GRAF_SLIDEBOX 142
GRAF_WATCHBOX 142
H. 52
HEAD 80
HERE 52
HEX 53
hexadecimal 53
HF8K.ACC 4
HIMEM 33
HOLD 53
HOME 53, 82
HSFORTH.PRG 4
I 53
IF 53
IKBDWS 91
INCLUDE 36
INITMOUSE 90
INTIN 94, 125
INTOUT 94, 125
introduction 7
IOREC 91
IS 53
J 53
JDISINT 91
JENABINT 91
K 53
KBDVBASE 92
KBRATE 92
KBSHIFT 89
KCLR 79
KERNEL.PRG 4
KEY 53
KEY.OFF 79
KEY.ON 79
KEYS.OFF 79
KEYS.ON 79
KEYTBL 91
L>NAME 53
LAST 66
LEAVE 53, 66
LFA 53
link field address 154
LINK> 53
LOAD 54
LOCAL 77
LOCATE 25
LOGBASE 90
LOGOUT 35, 54
LOMEM 33
LOOP 54, 66
LT 80
M* 54

M/ 54
M> 79
MACRO 54
MAKEFILE 35, 54
MALLOC 87
MAX 54
MEMORY 33
MENU_BAR 132
MENU_ICHECK 133
MENU_IENABLE 133
MENU_REGISTER 134
MENU_TEXT 134
MENU_TNORMAL 133
MFPINT 91
MFREE 87
MIDI 54
MIDI library **79**
MIDI.SEQ 4, 79
MIDIWS 91
MIN 54
MINUS 54
MOD 54, 65
MOFF 82
MON 82
MORE 35, 55
MULTI 78
Multi-Tasking **77**
MVTO 80
N>LINK 55
name field address 154
NAME> 55
NAND 55
NEGATE 55, 62
NFA 55, 154
NIP 55
NOR 55
NOS 8
NOT 55
NX 68
NXOR 55
O. 55
OBJC_ADD 134
OBJC_CHANGE 137
OBJC_DELETE 135
OBJC_DRAW 135
OBJC_EDIT 136
OBJC_FIND 135
OBJC_OFFSET 136
OBJC_ORDER 136
OBJECT.SEQ 4
OF 55
OFFGIBIT 92
OK 56
ONGIBIT 92
ONLY 56
OR 56
ORDER 56
OVER 56
PAD 56
PAGE 56
parameter field address 154
PAUSE 77
PB 93, 125
PD 80
PEN 80
PEXEC 83
PFA 56, 154
PHYSBASE 90
PICK 56, 62, 67
PIXEL 80
PLANES 81
PLOT 80
PNAME 56
POLY 81
PRETURN 83
PREVIOUS 47
PRG.SEQ 4
PRGINIT.SEQ 4
PRIMES.SEQ 4
PROGRAM.SEQ 4
PROTOBT 91
PRT 56
PRTBLK 92
PTERMRES 83
PTSIN 94
PTSOUT 94
PU 80
PUNTAES 92
QUIT 56
QX 68
R 57
R/W 57
R> 57
R@ 57, 62
RAM 33
RAMCLR 33
RAMDISK 33
RAMK 33
RANDOM 91
README File **5**
RECT 81
Registration Card **3**
REPEAT 57
ROLL 57, 62, 67
ROT 57
RP! 57, 67
RP@ 57
RSCONF 91
RSP 57
RSRC_FREE 150
RSRC_GADDR 150
RSRC_LOAD 150
RSRC_OBFIX 151
RSRC_SADDR 151

RT 80
Rubik's Cube 31
RWABS 89
S->D 57
S>D 67
SAVE-BUFFERS 57
SCAN 58
SCRDMP 91
SCRP_READ 144
SCRP_WRITE 144
SEAL 58, 68
SEQ>BLK 37
SETCOLOR 90
SETEXEC 89
SETPALLETE 90
SETPRT 92
SETSCREEN 90
SETTIME 91
SHEL_ENVRN 152
SHEL_FIND 152
SHEL_READ 151
SHEL_WRITE 152
SHRINK 68
SIGN 58
SINE 81
SINGLE 78
SLEEP 78
SLOT 68
SP! 58, 67
SP@ 58
SPACE 58
SPACES 58
SPAN 58
SPRITE 81
SSBRK 90
Stack 8
STATE 58
STOP 78
STRING$ 58
STRLEN 58
SUPER 85
SUPEREXEC 92
SWAP 58
SYSVAR 58
SYSVEC 68
TASK 78
Technical Support 157
TERM 83
TGETDATE 84
TGETTIME 85
THEN 58
THRU 58
TIB 59, 67
TOGGLE 59
TOS 8, 59
TSETDATE 84
TSETTIME 85
TUCK 59
TURTLE 80
TYPE 59
U* 59
U. 59
U.R 59
U/ 59
U/MOD 62
U< 71
UBIK 31
UBIK.SEQ 4
UM* 67
UM/MOD 67
UNDER 59
UNTIL 59
UNWRAP 81
UPC 59
UPDATE 59
Upgrades 157
UPPER 59
upper case 59
USER 77
V 78
V_ARC 102
V_BAR 101
V_CELLARRAY 100
V_CIRCLE 102
V_CLRWK 97
V_CLSVWK 97
V_CLSWK 97
V_CONTOURFILL 101
V_ELLARC 103
V_ELLIPSE 103
V_ELLPIE 103
V_ESCAPES 122
V_FILLAREA 99
V_GET_PIXEL 114
V_GTEXT 99
V_HIDE_C 118
V_JUSTIFIED 104
V_OPNVWK 97
V_OPNWK 94
V_PIE 102
V_PLINE 98
V_PMARKER 99
V_RBOX 104
V_RFBOX 104
V_SHOW_C 118
V_UPDWK 97
VARIABLE 59, 62
VDISYS 94
VERSION 85
VEX_BUTV 118
VEX_CURV 119
VEX_MOTV 119
VEX_TIMV 117
VLIST 69

VOC-LINK 60
VOCS 69
VQ_COLOR 120
VQ_EXTND 120
VQ_KEY_S 119
VQ_MOUSE 118
VQF_ATTRIBUTES 121
VQIN_MODE 122
VQL_ATTRIBUTES 120
VQM_ATTRIBUTES 121
VQT_ATTRIBUTES 121
VQT_EXTENT 121
VR_RECFL 101
VR_TRNFM 114
VRO_CPYFM 113
VRQ_CHOICE 116
VRQ_LOCATOR 115
VRQ_STRING 116
VRQ_VALUATOR 115
VRT_CPYFM 113
VS_CLIP 97
VS_COLOR 105
VSC_FORM 117
VSF_COLOR 112
VSF_INTERIOR 111
VSF_PERIMETER 112
VSF_STYLE 112
VSF_UDPAT 113
VSIN_MODE 115
VSL_COLOR 107
VSL_ENDS 107
VSL_TYPE 106
VSL_UDSTY 106
VSL_WIDTH 106
VSM_COLOR 108
VSM_HEIGHT 108
VSM_TYPE 107
VST_ALIGNMENT 111
VST_COLOR 110
VST_EFFECTS 110
VST_FONT 110
VST_HEIGHT 108
VST_POINT 109
VST_ROTATION 109
VSWR_MODE 105
VSYNC 92
W 41
W! 41
W* 60
W/ 60
W@ 45
WAKE 78
WARRAY 47
WHERE 25
WHILE 60
WIND_CALC 149
WIND_CLOSE 146
WIND_CREATE 145
WIND_DELETE 146
WIND_FIND 148
WIND_GET 147
WIND_OPEN 146
WIND_SET 148
WIND_UPDATE 149
WORD 60, 63
Words **7**, 60, 69
WR> 57
WRAP 81
WTOGGLE 60
XBTIMER 92
XOR 60
XSPRITE 81

*HiSoft FORTH for the Atari ST Computers*

HiSoft
High Quality Software

Produced in England
ISBN 0 948517 24 7